conference

proceedings

The Fourth Conference on Object-Oriented Technologies and Systems (COOTS) Proceedings

Santa Fe, New Mexico
April 27–30, 1998

Sponsored by
The USENIX Association

USENIX®

The Advanced Computing
Systems Association

USENIX Fourth Conference on Object-Oriented Technologies and Systems (COOTS) Santa Fe, New Mexico April 1998

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
WWW URL: http://www.usenix.org

The price is $22 for members and $32 for nonmembers.
Outside the U.S.A. and Canada, please add
$12 per copy for postage (via air printed matter).

Past COOTS Proceedings

COOTS III	June 1997	Portland, Oregon	$20/30
COOTS II	June 1996	Toronto, Canada	$20/30
COOTS I	June 1995	Monterey, CA	$18/24

ISBN 1-880446-93-6

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

Proceedings of the

Fourth USENIX Conference

on

Object-Oriented Technologies and Systems

(COOTS)

April 27-30, 1998
Santa Fe, New Mexico

Conference Organizers

Program Chair
Joe Sventek, Hewlett-Packard Company

Tutorial Program Chair
Douglas C. Schmidt, Washington University

Program Committee
Ken Arnold, Sun Microsystems, Inc.
Gerald Baumgartner, Ohio State University
David Chappell, Chappell & Associates
Murthy Devarakonda, IBM
Daniel Edelson, Ascend Communications
Jennifer Hamilton, Microsoft Corporation
Doug Lea, SUNY Oswego
Gary Leavens, Iowa State University
Dmitry Lenkov, Hewlett-Packard Company
Jeff Magee, Imperial College of Science, Technology & Medicine
Scott Meyers, Software Development Consultant
Louise Moser, UC Santa Barbara
Ira Pohl, UC Santa Cruz
Rajendra Raj, Morgan Stanley & Company
Doug Schmidt, Washington University
Steve Vinoski, IONA Technologies, Inc.

The USENIX Association Staff

Table of Contents

The Fourth USENIX Conference on Object-Oriented Technologies and Systems (COOTS)

April 27-30, 1998
Santa Fe, New Mexico

Wednesday, April 29, 1998

Thursday, April 30, 1998

Experience Papers
Session Chair: Douglas C. Schmidt, Washington University

Mobility
Session Chair: Doug Lea, State University of New York, Oswego

Programming Techniques
Session Chair: Jennifer Hamilton, Microsoft Corporation

Quality of Service Specification in Distributed Object Systems Design

Svend Frølund
Hewlett-Packard Laboratories, frolund@hpl.hp.com

Jari Koistinen
Hewlett-Packard Laboratories, jari@hpl.hp.com

Traditional object-oriented design methods deal with the functional aspects of systems, but they do not address quality of service (QoS) aspects such as reliability, availability, performance, security, and timing. However, deciding which QoS properties should be provided by individual system components is an important part of the design process. Different decisions are likely to result in different component implementations and system structures. Thus, decisions about component-level QoS should be made at design time, before the implementation is begun. Since these decisions are an important part of the design process, they should be captured as part of the design. We propose a general Quality-of-Service specification language, which we call QML. In this paper we show how QML can be used to capture QoS properties as part of designs. In addition, we extend UML, the de-facto standard object-oriented modeling language, to support the concepts of QML. QML is designed to integrate with object-oriented features, such as interfaces, classes, and inheritance. In particular, it allows specification of QoS properties through refinement of existing QoS specifications. Although we exemplify the use of QML to specify QoS properties within the categories of reliability and performance, QML can be used for specification within any QoS category—QoS categories are user-defined types in QML.

1. Introduction

1.1 Quality-of-Service in Software Design

In software engineering—like any engineering discipline—design is the activity that allows engineers to invent a solution to a problem. The input to the design activity consists of various requirements and constraints. The result of a design activity is a solution in which all major architectural and technical problems have been addressed. Design is an important activity since it allows engineers to invent solutions stepwise and in an organized manner. It makes engineers consider solutions and trade various system functions against each other.

To be useful, computer systems must deliver a certain *quality of service* (QoS) to its users. By QoS, we refer to non-functional properties such as performance, reliability, availability, and security. Although the delivered QoS is an essential aspect of a computer system, traditional design methods, such as [2, 11, 1, 13, 4], do not incorporate QoS considerations into the design process. We strongly believe that, in order to build systems that deliver their intended QoS, it is essential to systematically take QoS into account at design time, and not as an afterthought during implementation.

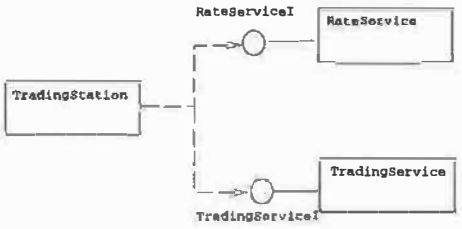


FIG. 1. Class diagram for the currency trading system

We use a simple example to illustrate the need for design-time QoS considerations. Consider the currency trading system in Figure 1. Currency traders interact with the `trading station`, which provides a user interface. To provide its functionality, the `trading station` uses a `rate service` and a `trading service`. The `rate service` provides rates, interests, and other information important to foreign exchange trading. The `trading service` provides the mechanism for making trades in a secure way. An inaccessible currency trading system might incur significant financial loss, therefore it is essential that the system is highly available.

It is important, at design time, to decide the QoS properties of individual system components. For example, we need to decide the availability properties of the `rate service`. We can decide that the `rate service` should be highly available so that the `trading station` can rely exclusively on it for rate information. Alternatively, we can decide that the `rate service` need *not* be highly available. If the `rate service` is not highly available, the `trading station` cannot rely exclusively on it, but must be prepared to continue operation if the `rate service` fails. To continue operation, the `trading station` could connect to an external rate service. As the example shows, different availability properties for the `rate service` can result in different system architectures. It is important to decide on particular QoS properties, and thereby chose a specific architecture, at design time.

Besides the system architecture, the choice of QoS properties for individual components also affects the implementation of components. For example, the `rate service` can be implemented as a single process or as a process pair, where the process-pair implementation provides higher availability. Different QoS properties are likely to require different implementations. Moreover, the QoS properties of a component may affect the implementation of its clients. For example, with a single-process implementation, the `trading station` may have to explicitly detect failures and restart the `rate service`, whereas with a process-pair implementation, failures may be completely masked for the `trading station`.

1.2 Quality-of-Service Specification

In the previous section we argued that QoS properties of individual components reflect important design decisions, and that we need describe these QoS properties as part of the design process. To capture component-level QoS properties, we introduce a language called QML (QoS Modeling Language).

Consider the CORBA IDL [17] interface definition for the `rate service` in Figure 2.

A rate service provides one operation for retrieving the latest exchange rates with respect to two currencies. The other operation performs an analysis and returns a forecast for the specified currency. The interface definition specifies the syntactic signature for a service but does not specify any semantics or non-functional aspects. In contrast, we concern ourselves with how to specify the required or provided QoS for servers implementing this interface.

QML has three main abstraction mechanisms for QoS specification: *contract type*, *contract*, and *profile*. QML allows us to define contract types that represent specific QoS aspects, such as performance or reliability. A contract type defines the *dimensions* that can be used to characterize a particular QoS aspect. A dimension has a domain of values that may be ordered. There are three kinds of domains: *set* domains, *enumerated* domains, and *numeric* domains. A contract is an instance of a contract type and represents a particular QoS specification. Finally, QML profiles associate contracts with interfaces, operations, operation arguments, and operation results.

The QML definitions in Figure 3 include two contract types `Reliability` and `Performance`. The reliability contract type defines three dimensions. The first one represents the number of failures per year. The keyword "decreasing" indicates that a smaller number of failures is better than a larger one. Time-to-repair (`TTR`) represents the time it takes to repair a service that has failed. Again, smaller values are better than larger ones. Finally, `availability` represents the probability that a service is available. In this case, larger values represent stronger constraints while smaller values represent lower probabilities and are therefore weaker.

```
interface RateServiceI {
  Rates latest(in Currency c1,in Currency c2)
    raises (InvalidC);
  Forecast analysis(in Currency c)
    raises (Failed);
};
```

FIG. 2. The `RateServiceI` interface

We also define a contract named `systemReliabilty` of type `Reliability`. The contract specifies constraints that can be associated with, for example, an operation. Since the contract is named it can be used in more than one profile. In this case, the contract specifies an upper

```
type Reliability = contract {
  numberOfFailures: decreasing numeric no/year;
  TTR: decreasing numeric sec;
  availability: increasing numeric;
};

type Performance = contract {
  delay: decreasing numeric msec;
  throughput: increasing numeric mb/sec;
};

systemReliability = Reliability contract {
  numberOfFailures < 10 no/year;
  TTR {
        percentile 100 < 2000;
        mean < 500;
        variance < 0.3
      };
  availability > 0.8;
};

rateServerProfile for RateServiceI = profile {
  require systemReliability;
  from latest require Performance contract {
    delay {
        percentile 50 < 10 msec;
        percentile 80 < 20 msec;
        percentile 100 < 40 msec;
        mean < 15 msec
        };
  };

  from analysis require Performance contract {
    delay < 4000 msec
  };
};
```

FIG. 3. Contracts and Profile for RateServiceI

bound on the allowed number of failures. It also specifies an upper bound, a mean, and a variance for `TTR`. Finally, it states that availability must always be greater than 0.8.

Next we introduce a profile called `rateServerProfile` that associates contracts with operations in the `RateServiceI` interface. The first requirement clause states that the server should satisfy the previously defined `systemReliability` contract. Since this requirement is not related to any particular operation, it is considered a default requirement and holds for every operation. Contracts for individual operations are allowed only to strengthen (refine) the default contract. In this profile there is no default performance contract; instead we associate individual performance contracts with the two operations of the `RateServiceI` interface. For `latest` we specify in detail the distribution of delays in percentiles, as well as a upper bound on the mean delay. For `analysis` we specify only an upper bound and can therefore use a slightly simpler syntactic construction for the expression. Since throughput is omitted for both operations, there are no requirements or guarantees with respect to this dimension.

We have now effectively specified reliability and performance requirements on any implementation of the `rateServiceI` interface. The specification is syntactically separate from the interface definition, allowing different `rateServiceI` servers to have different QoS characteristics.

QoS specifications can be used in many different situations. They can be used during the design of a system to understand the QoS requirements for individual components that enable the system as a whole to meet its QoS goals. Such design-time specification is the focus of this paper. QoS specifications can also be used to dynamically negotiate QoS agreements between clients and servers in distributed systems.

In negotiation it is essential that we can match offered and required QoS characteristics. As an example, satisfying the constraint "delay < 10 msec" implies that we also satisfy "delay < 20 msec." We want to enable automatic checking of such relations between any two QoS specifications. We call this procedure *conformance checking*, and it is supported by QML.

QML allows designers to specify QoS properties independently of how these properties can be implemented. For example, QML enables designers to specify a certain level of availability without reference to a particular high-availability mechanism such as primary-backup or active replication.

QML supports the specification of QoS properties in an object-oriented manner; it provides abstraction mechanisms that integrate with the usual object-oriented abstraction mechanisms such as classes, interfaces, and inheritance. Although QML is not tied to any particular design notation, we show how to integrate QML with UML [2], and we provide a graphical syntax for component-level QoS specifications.

QML is a general-purpose QoS specification language; it is not tied to any particular domain, such as real-time or multi-media systems, or to any particular QoS *category*, such as reliability or performance.

We organize the rest of this paper in the following way. In Section 2, we introduce our terminology for distributed object systems. We present the dimensions of reliability and performance that we use in Section 3. We describe QML in Section 4, and we explain its integration into UML in Section 5. We use QML and the UML extensions to specify the QoS properties of a computer-based telephony system in Section 6. The topic of Section 7 is related work, and Section 8 is a discussion of our approach. Finally, in Section 9, we draw our conclusions.

2. Our Terminology for Object-Oriented Systems

We assume that a system consists of a number of *services*. A service has a number of *clients* that rely on the service to get their work done. A client may itself provide service to other clients.

A service has a *service specification* and an implementation. A service specification describes what a service provides; a service implementation consists of a collection of software and hardware objects that collectively provide the specified service. For example, a name service maintains associations between names and objects. A name service can be replicated, that is, it can be implemented by a number of objects that each contain all the associations. It is important to notice that we consider a replicated name service as *one* logical entity even though it may be implemented by a collection of distributed objects.

A client uses a service through a *service reference*, or simply a *reference*. A reference is a handle that a client can use to issue service requests. A reference provides a client with a single access point, even to services that are implemented by multiple objects.

Traditionally, a service specification is a functional *interface* that lists the operations and attributes that clients can access; we extend this traditional notion of a service specification to also include a definition of the QoS provided by the service. The same service specification can be realized by multiple implementations, and the same collection of objects can implement multiple service specifications.

3. Selected Dimensions

To specify QoS properties in QML, we need a way to formally quantify the various aspects of QoS. A *QoS*

category denotes a specific non-functional characteristic of systems that we are interested in specifying. *Reliability*, *security*, and *performance* are examples of such categories. Each category consists of one or more *dimensions* that represent a metric for one aspect of the category. *Throughput* would be a dimension of the performance QoS category. We represent QoS categories and dimensions as user-defined types in QML.

To meaningfully characterize services with QoS categories we need valid dimensions. We are particularly interested in the dimensions that characterize services without exposing internal design and implementation details. Such dimensions enable the specification of QoS properties that are relevant and understandable for, in principle, any service regardless of implementation technology.

We describe a set of dimensions for reliability and performance. In [8] we have reviewed a variety of literature and systems on reliability including work by Gray et al. [7], Cristian [5], Reibman [14], Birman, [3], Maffeis [10], Littlewood [9], and others. As a result we propose the following dimensions for characterizing the reliability of distributed object services:

Name	Type
TTR	Time
TTF	Time
Availability	Probability
Continuous availability	Probability
Failure masking	**set** {failure, omission, response, value, state, timing, late, early}
Server failure	**enum** {halt, initialState, rollBack}
Operation semantics	**enum** {exactlyOnce, atLeastOnce, atMostOnce}
Rebinding policy	**enum** {rebind, noRebind}
Number of failures	Unsigned Integer
Data policy	**enum** {valid, notValid}

We use the measurable quantities of time to failure (`TTF`) and time to repair (`TTR`). `Availability` is the probability that a service is available when a client attempts to use it. Assume for example that service is down totally one week a year, then the availability would be 51/52, which is approximately 0.98. `Continuous availability` assesses the probability with which a client can access a service an infinite number of times during a particular time period. The service is expected not to fail and to retain all state information during this time period. We could for example require that a particular client can use a service for a 60 minute period without failure with a probability of 0.999. `Continuous availability` is different from `availability` in that it requires subsequent use of a service to succeed but only for a limited time period.

The `failure masking` dimension is used to describe what kind of failures a server may expose to its clients. A client must be able to detect and handle any kind of exposed failure. The above table lists the set of all possible failures that can be exposed by services in general. The QoS specification for a particular service will list the subset of failures exposed by that service.

We base our categorization of failure types—shown in Figure 4—on the work by Cristian [5]. If a service exposes `omission` failures, clients must be prepared to handle a situation where the service simply omits to respond to requests. If a service exposes `response` failures, it might respond with a faulty return value or an incorrect state transition. Finally, if the service exposes `timing` failures, it may respond in an untimely manner. Timing failures have two subtypes: `late` and `early` timing errors. Services can have any combination of failure masking characteristics.

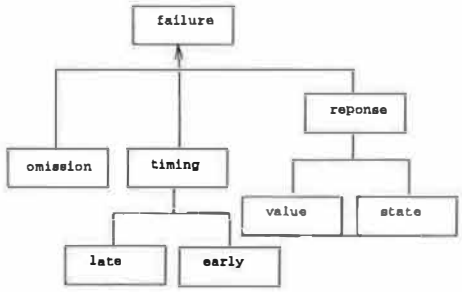


FIG. 4. Failure type hierarchy

`Operation semantics` describe how requests are handled in the case of a failure. We can specify that issued requests are executed `exactlyOnce`, `atLeastOnce`, or `atMostOnce`.

`Server failure` describes the way in which a service can fail. That is, whether it will `halt` indefinitely, restart in a well defined `initialState`, or restart `rolledBack` to a previous check point.

The `number of failures` gives a likely upper bound for the number of times the service will fail during a specific time period.

When a service fails the client needs to know whether it can use the existing reference or whether it needs to rebind to the service after the service has recovered. The `rebinding policy` is used to specify this aspect of reliability.

Finally, we propose that the client also needs to know if data returned by the service still is valid after the service has failed and been restarted. To specify this we need to associate `data policy` with entities such as return values and out arguments.

For the purpose of this paper we will propose a minimal set of dimensions for characterizing performance. We are only including `throughput` and `latency`. `Throughput` is the transfer rate for information, and can, for example, be specified as megabytes per second. `Latency` measures the time between the point that an in-

vocation was issued and the time at which the response was received by the client.

Dimensions such as those presented here constitute the vocabulary for QoS specification languages. We use the dimensions to describe the example in section 6.

4. QML: A Language to Specify QoS Properties

We describe the main design considerations for QML in Section 4.1. We already introduced the fundamental concepts of QML in section 1. Sections 4.2–4.8 describe the syntax and semantics of QML in more detail. For the full description of QML we refer to the language definition in [6].

4.1 Basic Requirements

The main design consideration for QML is to support QoS specification in an object-oriented context. We want QML to integrate seamlessly with existing object-oriented concepts. This overall goal results in the following specific design requirements for QML:

- QoS specifications should be syntactically separate from other parts of service specifications, such as interface definitions. This separation allows us to specify different QoS properties for different implementations of the same interface.
- It should be possible to specify both the QoS properties that clients require and the QoS properties that services provide. Moreover, these two aspects should be specified separately so that a client-server relationship has two QoS specifications: a specification that captures the client's requirements and a specification that captures the service's provisioning. This separation allow us to specify the QoS characteristics of a component, the QoS properties that it provides and requires, without specifying the interconnection of components. The separation is essential if we want to specify the QoS characteristics of components that are reused in many different contexts.
- There should be a way to determine whether the QoS specification for a service satisfies the QoS requirement of a client. This requirement is a consequence of the separate specification of the QoS properties that clients require and the QoS properties that services provide.
- QML should support refinement of QoS specifications. In distributed object systems, interface definitions are typically subject to inheritance. Since inheritance allows an interface to be defined as a refinement of another interface, and since we associate QoS specifications with interfaces, we need to support refinement of QoS specifications.
- It should be possible to specify QoS properties at a fine-grained level. As an example, performance characteristics are commonly specified for individual operations. As another example, the data policy dimension described in Section 3 is applicable to arguments and return values of operations. QML must allow QoS specifications for interfaces, operations, attributes, operation parameters, and operation results.

Other aspects such as negotiation and utility can be dealt with as mechanisms using QML or possibly be part of future extensions of QML. This paper focuses on the requirements listed above.

We have already briefly introduced the fundamental concepts of QML: *contract type*, *contract*, *profile*. The following sections will provide a more detailed description of QML.

4.2 Contracts and Contract Types

A contract type contains a *dimension type* for each of its dimensions. We use three different dimension types: set, enumeration, and numeric. Figure 5 gives an abstract syntax for contract and dimension types.

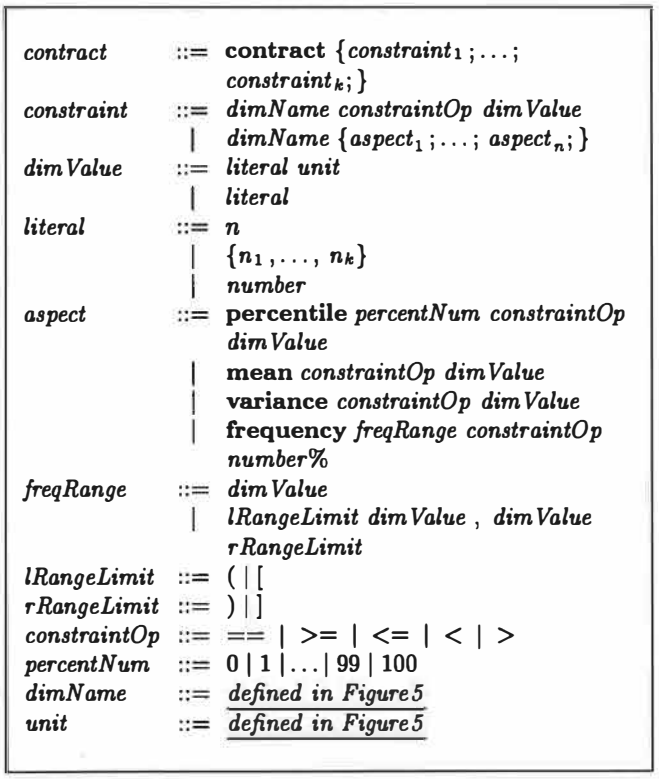
$$
\begin{array}{lcl}
\mathit{conType} & ::= & \textbf{contract}\ \{\mathit{dimName}_1 : \mathit{dimType}_1 ; \ldots ; \\
 & & \quad \mathit{dimName}_k : \mathit{dimType}_k ; \} \\
\mathit{dimName} & ::= & n \\
\mathit{dimType} & ::= & \mathit{dimSort} \\
 & | & \mathit{dimSort}\ \mathit{unit} \\
\mathit{dimSort} & ::= & \textbf{enum}\ \{n_1, \ldots, n_k\} \\
 & | & \mathit{relSem}\ \textbf{enum}\ \{n_1, \ldots, n_k\}\ \textbf{with}\ \mathit{order} \\
 & | & \textbf{set}\ \{n_1, \ldots, n_k\} \\
 & | & \mathit{relSem}\ \textbf{set}\ \{n_1, \ldots, n_k\} \\
 & | & \mathit{relSem}\ \textbf{set}\ \{n_1, \ldots, n_k\}\ \textbf{with}\ \mathit{order} \\
 & | & \mathit{relSem}\ \textbf{numeric} \\
\mathit{order} & ::= & \textbf{order}\ \{n_i < n_j, \ldots, n_k < n_m\} \\
\mathit{unit} & ::= & \mathit{unit/unit} \mid \% \mid \mathit{msec} \mid \ldots \\
\mathit{relSem} & ::= & \textbf{decreasing} \mid \textbf{increasing}
\end{array}
$$

FIG. 5. Abstract syntax for contract types

Contracts are instances of contract types. A contract type defines the structure of its instances. In general, a contract contains a list of constraints. Each constraint is associated with a dimension. For example, if we have a dimension "latency" in a contract type, a contract instance may contain the constraint "latency < 10." Figure 6 gives an abstract syntax for contracts and constraints.

A contract may specify constraints for all or a subset of the dimensions in its contract type. Omission of a specification for a particular dimension indicates that the contract is trivially satisfied along that dimension.

In general, a constraint consists of a name, an operator, and a value. The name is typically the name of a dimension, but, as we describe in Section 4.3, the name can also be the name of a dimension *aspect*. The permissible operators and values depend on the dimen-



contract	::=	**contract** {*constraint*$_1$; . . . ; *constraint*$_k$; }
constraint	::=	*dimName constraintOp dimValue*
	|	*dimName* {*aspect*$_1$; . . . ; *aspect*$_n$; }
dimValue	::=	*literal unit*
	|	*literal*
literal	::=	*n*
	|	{n_1 , . . . , n_k}
	|	*number*
aspect	::=	**percentile** *percentNum constraintOp dimValue*
	|	**mean** *constraintOp dimValue*
	|	**variance** *constraintOp dimValue*
	|	**frequency** *freqRange constraintOp number%*
freqRange	::=	*dimValue*
	|	*lRangeLimit dimValue* , *dimValue rRangeLimit*
lRangeLimit	::=	(| [
rRangeLimit	::=	) |]
constraintOp	::=	== | >= | <= | < | >
percentNum	::=	0 | 1 | . . . | 99 | 100
dimName	::=	*defined in Figure 5*
unit	::=	*defined in Figure 5*

FIG. 6. Abstract syntax for contracts

sion type. A dimension type specifies a domain of values. These values can be used in constraints for that dimension. The domain may be ordered. For example, a numeric domain comes with a built-in ordering ("<") that corresponds to the usual ordering on numbers. Set and enumeration domains do not come with a built-in ordering; for those types of domains we have to describe a user-defined ordering of the domain elements. The domain ordering determines which operators can be used in constraints for that domain. For example, we cannot use inequality operators ("<," ">," "<=," ">=") in conjunction with an unordered domain.

The domain for a set dimension contains elements that are sets of name literals. We specify a set domain using the keyword **set**, as in "**set** $\{n_1, \ldots, n_k\}$." This defines a set domain where the domain elements are subsets of the set "$\{n_1, \ldots, n_k\}$." The constraints over a set dimension will then be constraints with set values, as in "`failures` == {`response`, `omission`}."

The domain for an enumeration dimension contains elements that are name literals. We specify an enumeration domain using the keyword **enum**. For example, we could define an enumeration domain as follows: "**enum** $\{n_1, \ldots, n_k\}$." Here, the domain will contain the name literals "$n_1, \ldots, n_k$," and we can specify constraints as "`dataPolicy` == `valid`."

The domain of a numeric dimension contains elements that are real numbers. Constraints for a numeric dimension are written as "`latency` < 10."

Elements of numeric dimensions are always ordered. We can specify a user-defined ordering for set and enumerated dimensions in the following way: "**order** {`valid` < `invalid`}." When dimensions are ordered we need to specify whether larger or smaller values are considered stronger. As an example consider the dimension of availability. A larger numeric value for availability is a stronger that a smaller, we say that availability is an "increasing" dimension. Other dimensions, such as delay, are "decreasing" since smaller values are consider as stronger guarantees. Consequently, QML requires that we define ordered dimensions as either decreasing or increasing. For the data validity enum decreasing semantics seems most intuitive, since `valid` also satisfies `invalid`.

The example in Figure 7 gives an example of a contract type expression followed by a contract expression. Note that the contract expression is explicitly typed with a contract type name, this explicit typing enables the QML compiler to determine a unique contract type for any contract expression. So far we have only covered the syntax for contract values and contract types. In Section 4.4, we describe how to name contract values and contract types, and how to use those names in contract expressions.

4.3 Aspects

In addition to simple constraints QML supports more complex characterizations that are called *aspects*. An aspect is a statistical characterization; QML currently includes four generally applicable aspects: *percentile*, *mean*, *variance*, and *frequency*. Aspects are used for characterizations of measured values over some time period.

The percentile aspect defines an upper or lower value for a percentile of the measured entities. The statement **percentile** P denotes the strongest P percent of the measurements or occurrences that have been observed.

```
type T = contract { // A contract type expression
  s1: decreasing set { e1, e2, e3, e4 }
        with order {e2<e1, e1<e3, e3<e4};
  e1: increasing enum { a1, a2, a3 }
        with order {a2<a1, a3<a2};
  n1: increasing numeric mb / sec;
};

T contract { // A contract expression of type T
  s1 <= { e1, e2 };
  e1 < a2;
  n1 < 23;
};
```

FIG. 7. Example contract type and contract expressions

The aspect "**percentile** 80 < 6" states that the $80th$ percentile of measurements for the dimensions must be less than 6. We allow a constraint for a dimension to contain more than one percentile aspect, as long as the same percentile P does not occur more than once.

QML also allows the specification of frequency constraints for individual values which is useful with enumerated types, and for ranges, which is useful with numeric dimensions. Rather than specifying specific numbers for the frequency, QML allows us to specify the relative percentage with which values in a certain range occur. The constraint "**frequency** V > 20%" means that in more that 20% of the occurrences we should have the value V. The literal V can be a single value or if the dimension has an ordering, and only then, it may be a range. The constraint "**frequency** $[1, 3)$ > 35%" means that we expect 35% of the actual occurrences to be larger than 1 and less than or equal to 3.

Figure 8 shows some examples of aspects in contract expressions. The contract expression is preceded by the name of its corresponding contract type. For `s1` we define one constraint for the 20th percentile. The meaning of this is that the strongest 20% of the value must be less than the specified set value.

```
contractTypeName contract {
  s1 { percentile 20 < { e1, e2 }};
  e1 {
       frequency a1 <= 10 %;
       frequency a2 >= 80 %;
     };
  n1 {
       percentile 10 < 20;
       percentile 50 < 45;
       percentile 90 < 85;
       percentile 100 <= 120;
       mean >= 60;
       variance < 0.6;
     };
};
```

FIG. 8. Example contract expression

```
decl        ::= conTypeDecl | conDecl
conTypeDecl ::= type y = conType
conDecl     ::= x_c = conExp
conExp      ::= y contract
              | x_c refined by
                {constraint_1 ; ... ; constraint_k ;}
conType     ::= defined in Figure5
contract    ::= defined in Figure6
constraint  ::= defined in Figure6
```

FIG. 9. Abstract syntax for definition of contracts and contract types

For `e1` we define the frequencies that we expect for various values. For the value `a1` we expect a frequency of less than or equal to 10%. For `a2` we expect a frequency greater than or equal to 80%, and so forth.

The constraint on `n1` defines bounds for values in different percentiles over the measurements of `n1`. In addition, we define an upper bound for the mean and the variance.

4.4 Definition of Contracts and Contract Types

The definition of a contract type binds a name to a contract type; the definition of a contract binds a name to the value of a contract expression. Figure 9 illustrates the abstract syntax to define contracts and contract types. In the astract syntax, we use x_c as a generic name for contracts and y as a generic name for contract types.

We can define a contract B to be a refinement of another contract A using the construct "B = A **refined by**{...}" where A is the name of a previously defined contract. The contract that is enclosed by curly brackets ({...}) is a "delta" that describes the difference between the contracts A and B. We say that the delta *refines* A and that B is a *refinement* of A. The delta can specify QoS properties along dimensions for which specification was omitted in A. Furthermore, the delta can replace specifications in A with stronger specifications. The notion of "stronger than" is given by a conformance relation on constraints. We describe conformance in more detail in Section 4.8.

Figure 10 and Figure 11 illustrates how a named contract type (`Reliability`) can be define and how contracts of that type can be defined respectively. The contract type `Reliability` has the dimensions that we have identified within the QoS category of reliability described in section 3

The contract `systemReliability` is an instance of `Reliability`; it captures a system wide property, namely that operation invocation has "exactly once" (or transactional) semantics. The `systemReliability` only provides a guarantee about the invocation semantics, and does not provide any guarantees for the other dimensions specified in the `Reliability` contract type.

The contract `nameServerReliability` is defined as a refinement of another contract, namely the contract bound to the name `systemReliability`. In the example, we strengthen the `systemReliability` contract by providing a specification along the `serverFailure` dimension, which was left unspecified in the `systemReliability` contract.

```
type Reliability = contract {
  failureMasking: decreasing
    set {omission, lostResponse, noExecution,
      response, responseValue, stateTransition};
  serverFailure:
    enum {halt,  initialState, rolledBack};
  operationSemantics: decreasing
    enum {atLeastOnce, atMostOnce, once} with
    order {once < atLeastOnce, once < atMostOnce};
  rebindingPolicy: decreasing
    enum {rebind, noRebind} with
    order {noRebind < rebind};
  dataPolicy: decreasing enum {valid, invalid}
     with order {valid < invalid};
  numOfFailure: decreasing numeric failures/year;
  MTTR: decreasing numeric sec;
  MTTF: increasing numeric day;
  reliability: increasing numeric;
  availability: increasing numeric;
};
```

FIG. 10. Example contract type definition

```
systemReliability = Reliability contract {
  operationSemantics == once;
};

nameServerReliability = systemReliability {
  serverFailure == rolledBack;
};

type Performance = contract {
  latency: decreasing numeric msec;
  throughput: increasing numeric kb/sec;
};

traderResponse = Performance contract {
  latency { percentile 90 < 50 msec };
};
```

FIG. 11. Example contract definitions

4.5 Profiles

According to our definition, a service specification contains an interface and a QoS profile. The interface describes the operations and attributes exported by a service; the profile describes the QoS properties of the service. A profile is defined relative to a specific interface, and it specifies QoS contracts for the attributes and operations described in the interface. We can define multiple profiles for the same interface, which is necessary since the same interface can for example have multiple implementations with different QoS properties.

Once defined, a profile can be used in two contexts: to specify client QoS requirements and to specify service QoS provisioning. Both contexts involve a ***binding*** between a profile and some other entity. In the client context this other entity is the service reference used by the client; in the service context, the entity is a service implementation. We discuss bindings in Section 4.7. Here, we describe a syntax for profile values, and in Section 4.6 we describe a syntax for profile definition.

profile	::=	**profile** $\{req_1; \ldots; req_n;\}$
req	::=	**require** *contractList*
	|	**from** *entityList* **require** *contractList*
contractList	::=	$conExp_1, \ldots, conExp_n$
entityList	::=	$entity_1, \ldots, entity_n$
entity	::=	*opName*
	|	*attrName*
	|	*opName.parName*
	|	**result of** *opName*
opName	::=	*identifier*
attrName	::=	*identifier*
parName	::=	*identifier*
conExp	::=	*defined in Figure 9*

FIG. 12. Abstract syntax for profiles

Figure 12 gives an abstract syntax for profiles. A profile is a list of requirements, where a requirement specifies one or more contracts for one or more interface entities, such as operations, attributes, or operation parameters. If a requirement is stated without an associated entity, the requirement is a ***default requirement*** that applies by default to all entities within the interface in question. Our intention is that the default contract is the strongest contract that applies to ***all*** entities within an interface. We can then explicitly specify a stronger contract for individual entities by using the refinement mechanism.

Contracts for individual entities are defined as follows: "**from** e **require** C." Here e is an entity and C is a contract. We use C as a delta that refines the default contract of the enclosing profile. Using individual entity contracts as deltas for refinement means that we do not have to repeat the default QoS constraints as part of each individual contract.

declaration	::=	*conTypeDecl*
	|	*conDecl*
	|	*profileDecl*
profileDecl	::=	x_p **for** *intName* = *profileExp*
profileExp	::=	*profile*
	|	x_p **refined by** $\{req_1; \ldots; req_n;\}$
intName	::=	*identifier*
conTypeDecl	::=	*defined in Figure 9*
conDecl	::=	*defined in Figure 9*
profile	::=	*defined in Figure 12*
req	::=	*defined in Figure 12*

FIG. 13. Abstract syntax for definition of profiles

Although a profile refers to specific operations and arguments within an interface, the final association between the profile and the interface is established in a profile definition. Such definitions are described in section 4.6.

For each contract type, such as reliability, that a profile involves, we may specify zero or one default contract. In addition, at most one contract of a given type can be explicitly associated with an interface entity.

If, for a given contract type T, there is no default contract and there is no explicit specification for a particular interface entity, the semantics is that no QoS properties within the category of T are associated with that entity.

4.6 Definition of Profiles

A profile definition associates a profile with an interface and gives the profile a name. A general requirement is that the interface entities referred to by the profile must exist in the related interface. The syntax for profile definition is given in Figure 13. The definition "*id* **for** *intName* = *prof*" gives the name *id* to the profile which is the result of evaluating the profile expression *prof* with respect to the interface *intName*. The profile name can be used to associate this particular profile with implementations of the *intName* interface or with references to objects of type *intName*.

A profile expression (*profileExp*) can be a profile, or an identifier with a "{...}" clause. If the expression is a profile value, the definition binds a name to this value. If a profile expression contains an identifier and a "{...}" clause, the identifier must be the name of a profile, and the "{...}" clause then refines this profile. The definition gives a name to this refined profile.

If we have a profile expression "*A* **refined by** {...}," then the delta must either add to the specifications in *A* or make the specifications in *A* stronger. The delta can add specifications by defining individual contracts for entities that do not have individual contracts in *A*. Moreover, the delta can specify a default contract if no default contract is specified in *A*. The delta can strengthen *A*'s specifications by giving individual contracts for entities that also have an individual contract in *A*. The individual contract in the delta are then used as a contract delta to refine the individual contract in *A*. Similarly, the delta can specify a contract delta that refines the default contract in *A*. We give a more detailed and formal description of profile refinement in [6].

```
interface NameServer {
  void init();
  void register(in string name, in object ref);
  object lookup(in string name);
}

nameServerProfile for NameServer = profile {
  require nameServerReliability;
  from lookup require Reliability contract {
    rebindPolicy == noRebind;
  };
}
```

FIG. 14. The interface of a name server

To exemplify the notion of profile definition, consider the interface of a name server in Figure 14. The profile called `nameServerProfile` is a profile for the `NameServer` interface; it associates various contracts with the operations defined with the `NameServer` interface. The `nameServerProfile` associates the `nameServerReliability` contract (introduced in Figure 11) as the default contract, and it associates a refinement of the `nameServerReliability` contract with the `lookup` operation.

Notice that the contract for the `lookup` operation must refine the default contract (in this case, the default contract is `nameServerReliability`). Since the contract for operations must always refine the default contract, it is implicitly understood that the contract expression in an operation contract is in fact a refinement.

4.7 Bindings

There are many ways in which QoS profiles can be bound to specific services. They can be negotiated and associated with deals between clients and server, or they can be associated statically at design or deployment time. For the purpose of this paper we will provide an example binding mechanism that allows clients to statically bind profiles to references. In addition, we allow a server to state the profile of its implementation. These bindings could be used to ensure compatible characteristics for clients and servers as well as runtime monitoring. An abstract syntax for our notion of binding is illustrated in Figure 16.

Figure 17 illustrates our notion of binding. In the first example the client declares a reference called `myNameServer` as a reference to a name server. The client's QoS requirements are expressed by means of the profile called `nameServerProfile`. In the second example, the implementation called `myNameServerImp` is declared to implement the service specification that consists of the interface called `NameServer` and the profile called `nameServerProfile`.

The binding mechanism need not be a part of QML but has been included here for clarity. Bindings are more closely related to interface specification, design and implementation languages. As an example we will propose a binding mechanism for UML in section 5.

4.8 Conformance

We define a conformance relation on profiles, contracts, and constraints. A stronger specification conforms to a weaker specification. We need conformance at runtime so that client-server connections do not have to be based on exact match of QoS requirements with QoS properties. Instead of exact match, we want to allow a service to provide more than what is required by a client. Thus, we want service specifications to conform to client specifications rather than match them exactly.

Profile conformance is defined in terms of contract conformance. Essentially, a profile P conforms to another profile Q if the contracts in P associated with an entity e conform to the contracts associated with e in the profile Q.

Contract conformance is in turn defined in terms of conformance for constraints. Constraint conformance defines when one constraint in a contract can be considered stronger, or as strong as, another constraint for the same dimension in another contract of the same contract type.

To determine constraint conformance for set dimensions, we need to determine whether one subset conforms to another subset. Conformance between two subsets depends on their ordering. In some cases, a subset represents a stronger commitment than its supersets. As an example, let us consider the failure-masking dimension. If a value of a failure-masking dimension defines the failures exposed by a server, a subset is a stronger commitment than its supersets (the fewer failure types exposed, the better). If, on the other hand, we consider a payment protocol dimension for which sets represent payment protocols supported by a server, a superset is obviously a stronger commitment than any of its subsets (the more protocols supported, the better). Thus,

```
binding          ::= clientBinding
                   | serviceBinding
clientBinding    ::= refDecl with profileExp
serviceBinding   ::= serviceDecl with profileExp
refDecl          ::= identifier : intName
serviceDecl      ::= identifier implements intName
```

FIG. 16. Abstract syntax for bindings

```
//Client side binding
myNameServer: NameServer with nameServerProfile;

//Implementation binding
myNameServerImp implements NameServer
  with nameServerProfile;
```

FIG. 17. Example bindings

to be able to compare contracts of the same type the dimension declarations need to define whether subsets or supersets are stronger.

A similar discussion applies to the numeric domain. Sometimes, larger numeric values are considered conceptually stronger than smaller. As an example, think of throughput. For dimensions such as latency, smaller numbers represent stronger commitments than larger numbers.

In general, we need to specify whether smaller domain elements are stronger than or weaker than larger domain elements. The **decreasing** declaration implies that smaller elements are stronger than larger elements. The **increasing** declaration means that larger elements are stronger than smaller elements. If a dimension is declared as decreasing, we map "stronger than" to "less than" (<). Thus, a value is stronger than another value, if it is smaller. An increasing dimension maps "stronger than" to "greater than" (>). The semantics will be that larger values are, considered stronger.

We want conformance to correspond to constraint satisfaction. For example, we want the constraint $d < 10$ to conform to the constraint $d < 20$. But $d < 10$ only conforms to $d < 20$ if the domain is decreasing (smaller values are stronger). To achieve the property that conformance corresponds to constraint satisfaction, we allow only the operators $\{==, <=, <\}$ for decreasing domains, and we allow only the operators $\{==, >=, >\}$ for increasing domains. Thus, if we have an increasing domain, the constraint $d < 20$ would be illegal.

If a profile Q is a refinement of another profile P, Q will also conform to P. Refinement is a static operation that gives a convenient way to write QoS specifications in an incremental manner. Conformance is a dynamic operation that, at runtime, can determine whether one specification is stronger than another specification. For more details on conformance we refer to [6].

5. An Extension of the Unified Modeling Language

In order to make QoS considerations an integral part of the design process, design notations must provide the appropriate language concepts. We have already presented a textual syntax to define QoS properties. Here, we extend UML [2] to support the definition of QoS properties. Later, we will use CORBA IDL [17] and our extension of UML [2] to describe an example design that includes QoS specifications.

In UML, *classes* are represented by rectangles. In addition, UML has a *type* concept that is used to describe abstractions without providing an implementation. Types are drawn as classes with a type stereotype annotation added to it. In UML, classes may implement types. The UML *interface* concept is a specialized usage of types. Interfaces can be drawn as small circles

that can be connected to class symbols. A class can use or provide a service specified by an interface. The example below shows a client using (dotted arrow) a service specified by an interface called I. We also show a class `Implementation` implementing the I interface but in this example the interface circle has been expanded to a class symbol with the *type* annotation.

Our extension to UML allows QoS profiles to be associated with *uses* and *implements* relationships between classes and interfaces. A reference to a profile is drawn as a rectangle with a dotted border within which the profile name is written. This profile box is then associated with a uses or implements relationship.

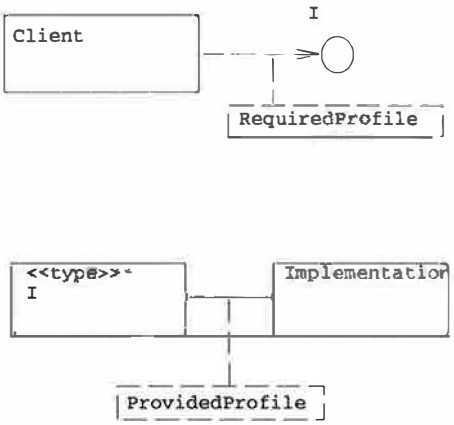


FIG. 15. UML extensions

In example 15, the client requires a server that implements the interface I and satisfies the QoS requirements stated in the associated `RequiredProfile`. The `Implementation` on the other hand promises to implement interface I with the QoS properties defined by the `ProvidedProfile` profile. The profiles are defined textually using our QoS specification language.

Our UML extension allows object-oriented design to be annotated with profile names that refer to separately defined QoS profiles. Notice that our UML extension associates profiles with specific implementations and usages of interfaces. This allows different clients of the same interface to require different QoS properties, and it allows different implementations of the same interface to provide different QoS properties.

6. Example

To illustrate QML and demonstrate its utility, we use it to specify the QoS properties of an example system. The example shows how QML can help designers decompose application level QoS requirements into QoS properties for application components. The example also demonstrates that different QoS trade-offs can give rise to different designs.

This example is a simplified version of a system for executing telephony services, such as telephone banking, ordering, etc. The purpose of having such an execution system is to allow rapid development and installation of new telephony services. The system must be scalable in order to be useful both in small businesses and for servicing several hundred simultaneous calls. More importantly—especially from the perspective of this paper—the system needs to provide services with sufficient availability.

Executing a service typically involves playing messages for the caller, reacting to key strokes, recording responses, retrieving and updating databases, etc. It should be possible to dynamically install new telephone services and upgrade them at runtime without shutting down the system. The system answers incoming telephone calls and selects a service based on the phone number that was called. The executed service may, for example, play messages for the caller and react to events from the caller or events from resources allocated to handle the call.

Telephone users generally expect plain old telephony to be reliable, and they commonly have the same expectations for telephony services. A telephony service that is unavailable will have a severe impact on customer satisfaction, in addition, the service company will loose business. Consequently, the system needs to be highly available.

Following the categorization by Gray et al. [7], we want the telephony service to be a *highly-available* system which means it should have a total maximum downtime of 5 minutes per year. The availability measure will then be 0.99999. We assume the system is built on a general purpose computer platform with specialized computer telephony hardware. The system is built using a CORBA [17] Object Request Broker (ORB) to achieve scalability and reliability through distribution.

6.1 System Architecture

We call the service execution system module `PhoneServiceSystem`. As illustrated by Figure 18, it uses an `EventSystem` module and a `TraderService` module.

Opening up the `PhoneServiceSystem` module in Figure 19, we see its main classes and interfaces. Classes are drawn as rectangles and interfaces as circles. Classes implement and use interfaces. As an example, the diagram shows that `ServiceExecutor` implements `ServiceI` and uses `TraderI`. In the diagram we have included references to QML profiles—such as `PlayerProfile_P`—of which a subset will be described in section 6.2. To ease the reading of the diagram we have named *required* and

provided profiles so that they end with the letters *R* and *P* respectively. We have omitted to draw some interrelationships for the purpose of keeping the diagram simple.

`CallHandlerI`, `ServiceI`, and `ResourceI` are three important interfaces of the system. The model also shows that the system uses interfaces provided by the `EventService` and `TraderService`.

When a call is made, the `CallHandlerImpl` receives the incoming call through the `CallHandlerI` interface and invokes the `ServiceExecutor` through the `ServiceI` interface. `CallHandlerImpl` receives the telephone number as an argument and maps that to a service identity. When `CallHandlerImpl` calls the `ServiceExecutor` it supplies the service identifier as an argument and a `CallHandle`. The `CallHandle` contains information about the call—such as the speech channel—that is needed during the execution of the service. A new instance of `CallHandle` is created and initialized by the `CallHandler` when an incoming call is received. The information in the `CallHandle` remains unchanged for the remainder of the call.

In order to execute a service, the `ServiceExecutor` retrieves the service description associated with the received service identifier. It also needs to allocate resources such as databases, players, recorders, etc. To obtain resources, the `ServiceExecutor` calls the `Trader`. Each resource offer its services when it is initially started by contacting the trader and registering its offer. To reduce complexity of the diagram we omit showing that resources use the trader.

`ServiceExecutor` uses the `PushSupplier` and implements the `PushConsumer` interface in the `EventService` module. Resources connect to the event service by using the `PushConsumer` interfaces. The communication between the service executor and its resources is asynchronous. When the service executor needs a resource to perform an operation, it invokes the resource which returns immediately. The service executor will then continue executing the service or stop to wait for events. When the resource has finished its operation, it notifies the service executor by sending an event through the event service. This communication model allows the service executor to listen for events from many sources at the same time, which is essential if, for example, the service executor simultaneously initiates the playing of menu alternatives and waits for responses from the caller.

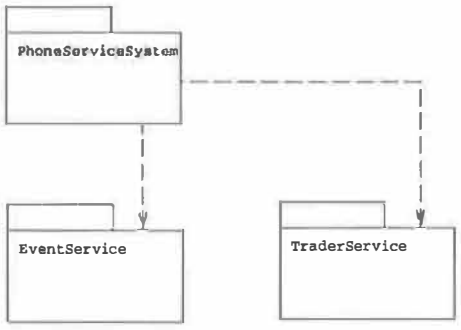


FIG. 18. High-level architecture

Figure 19 also includes references to QoS profiles. In new designs, clients and services are usually designed to match each others needs therefore the same profile often specifies both what clients expect and what services provide. When clients and services refer to the same profiles, it becomes trivial to ensure that the requirements by a client are satisfied by the service. To point out an example, `CallHandlerImpl` requires that the `ServiceI` interface is implemented with the QoS properties defined by `SEProfile_P` and at the same time `ServiceExecutor` provides `ServiceI` according to the same QoS profile.

In other cases, such as the `Trader`, are expected to preexist and therefore have previously specified QoS properties. In those situations we have one contract specifying the required properties and another contract specifying what is provided. Consequently we need to make sure the provided characteristics satisfy the required; this is referred to as *conformance* and is discussed in section 4.8.

We will now present simplified versions of three main interfaces in the design. The `ServiceI` interface provides an operation, called `execute`, to start the execution of a service. The service identifier is obtained from a table that maps phone numbers to services. The `CallHandle` argument contain channel identifiers and other data necessary to execute the service.

The `Trader` allows resources to offer and withdraw their services. Service executors can invoke the `find` or `findAll` operations on the `Trader` to locate the resources they need. Using a trader allows us to decouple `ServiceExecutors` and resources. This decoupling make it possible to smoothly introduce new resources and remove malfunctioning or deprecated resources. Observe that this is a much simplified trader for the purpose of this paper.

Finally, we have the `PlayerI` that represents a simple player resource. Players allow us to play a sequence of messages on the connection associated with the supplied `CallHandle`. The idea is that a complete message can be built up by a sequence of smaller phrases. The interface allows the service executor to interrupt the playing of messages by calling `stop`.

6.2 Reliability

We have already shown in Figure 19 how profiles are associated with *uses* and *implements* relationships between interfaces and classes. We will now in more depth discuss what the QoS profiles and contracts should be for this particular design. For the contracts we will use the dimensions proposed in section 3. We will not

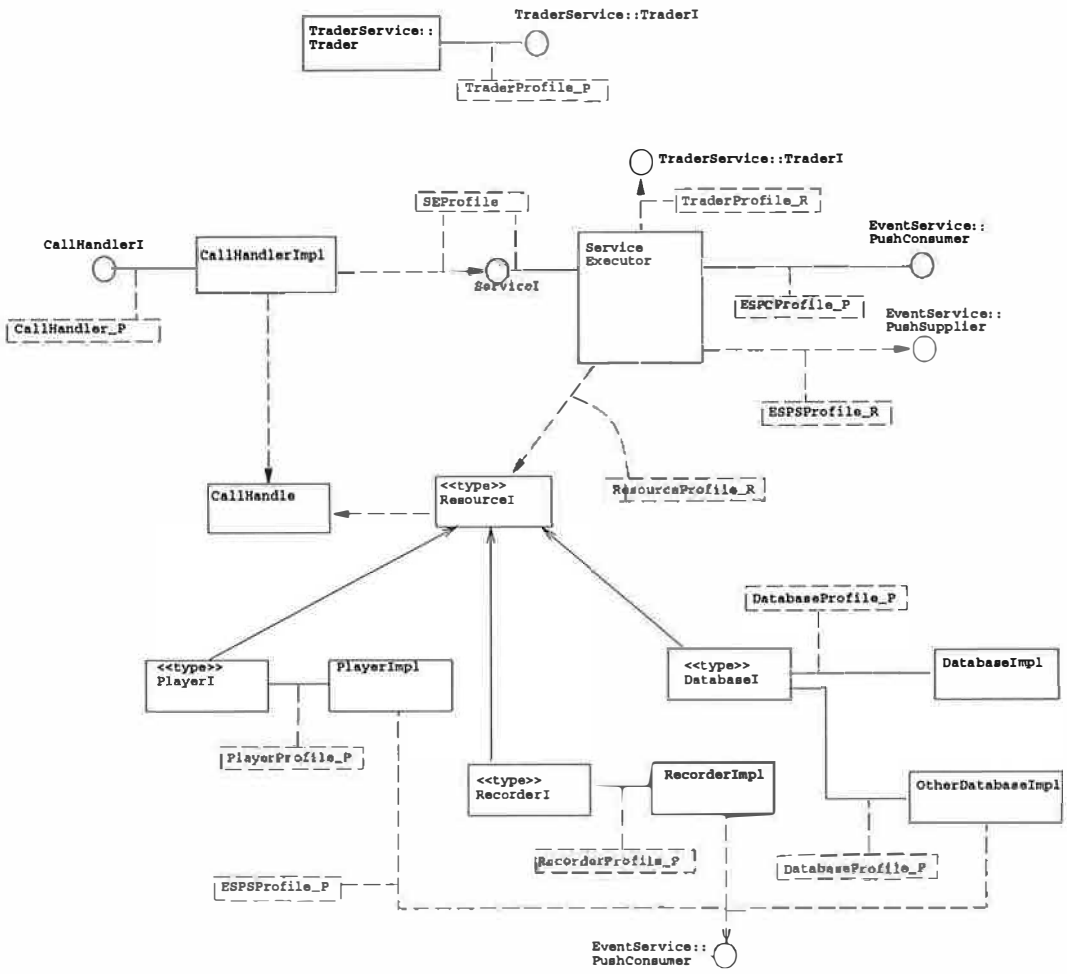


FIG. 19. Class diagram for PhoneServiceSystem

present any development process with which you identify important profiles and their content.

To meet end-to-end reliability requirements, the underlying communications infrastructure, as well as the execution system, must meet reliability expectations. We assume that the communications infrastructure is reliable, and focus on the reliability of the service execution system.

From a telephone user's perspective, the interface `CallHandlerI` represents the peer on the other side of the line. Thus, to provide high-availability to telephone users, the `CallHandlerI` service must be highly-available.

To provide a highly-available telephone service, we require that the `CallhandlerImpl` has very short recovery time and long time between failures. Due to the expected shopping behavior of telephone service users we must require the repair time (`MTTR`) to not significantly exceed 2 minutes and that the variance is small.

The `CallHandler` does not provide any sophisticated failure masking, but it has a special kind of object reference that does not require rebinding after a failure. We

```
interface ServiceI {
  void execute(in ServiceId si, in CallHandle ch)
    raises (InvalidSI);
  boolean probe() raises (ProbeFailed);
};
```

FIG. 20. The ServiceI interface

```
interface TraderI {
  OfferId offer(in OfferRec or, in Object obj)
    raises (invalidOffer);
  Match find(in Criteria cr) raises (noMatch);
  MatchSeq findAll(in Criteria cr) raises (noMatch);
  void withdraw(in OfferId o) raises (noMatch);
};
```

FIG. 21. The TraderI interface

```
CallServerReliability = Reliability contract {
  MTTR {
         percentile 100 <= 2;
         variance <= 0.3
       };
  TTF {
         percentile 100 > 0.05 days;
         percentile 80 > 100 days;
         mean >= 140 days;
      };
  availability >= 0.99999;
  contAvailability >= 0.99999;
  failureMasking ==  { omission };
  serverFailure == initialState;
  rebindPolicy == noRebind;
  numOfFailure <= 2  failures/year;
  operationSemantics == atMostOnce;
};

CallHandlerProfile_P for CallHandlerI = profile {
  require CallServerReliability;
}
```

FIG. 22. Contract and binding for CallHandler

are prepared to accept on average 2 failures per year. If the service fails, any executing and pending requests are discontinued and removed. This means we have a **at most once** operation semantics. The contract and profile of `CallHandlerI` as provided by `CallHandlerImpl` is described in Figure 22.

From Figure 19 we can see that the reliability of `CallHandlerI` directly depends on the reliability of service defined by `ServiceI`. `ServiceExecutor` can not provide any services without resources. Unless `ServiceExecutor` can handle failing traders and resources the reliability depends directly on the reliability of `TraderI` and any resources it uses. In this example we want to keep the `ServiceExecutor` as small and simple as possible, therefore we propagate high-availability requirements from `CallHandlerI` to the trader and the resources. This is certainly a major design decision which will affect the design and implementation of the other components of the system.

We expect the `ServiceExecutor` to have a short recovery time since it holds no information that we wish to recover. If it fails, the service interactions it currently executes will be discontinued. We assume that users consider it more annoying if a session is interrupted due to a failure than if they are unable to connect to the service. We therefore require the `ServiceExecutor` to be reliable in the sense that it should function adequately over the duration of a typical service call. Calls are estimated to last 3 minutes on average with 80% of the calls less than 5 minutes. With this in mind, we will require that the service executor provides high continuous availability with a time period of 5 minutes.

Since the recovery time is short, we can allow more frequent failures without compromising the availability requirements.

The `ServiceExecutor` recovers to a well defined **initial state** and will forget about all executions that where going on at the time of the failure. The contract states that rebinding is necessary, which means that when the service executor is restarted, the `CallHandler` receives a notification that it can obtain a reference to the `ServiceExecutor` by rebinding. Pending requests are executed at most once in case of a failure; most likely they are not executed at all which is considered acceptable for this system. The contract and profile used for `ServiceI` are described in Figure 24.

Although the `ServiceExecutor` itself can recover rapidly, it still depends on the `Trader` and the resources.

We expect the `Trader` to have a relatively short recovery time, which relaxes the mean time to failure requirements slightly. We insist that all types of telephony

```
interface PlayerI : ResourceI {
  void play(in CallHandle ch, in MsgSeq ms)
    raises (InvalidMsg);
  void stop(in CallHandle ch);
};
```

FIG. 23. The PlayerI interface

```
ServiceExecutorReliability = Reliability contract
{
  MTTR < 20 sec;
  TTF {
         percentile 100 > 0.05 days;
         percentile 80 > 20 days;
         mean > 24 days;
       }
  availability >= 0.99999;
  contAvailability > 0.999999 ;
  failureMasking ==  { omission };
  serverFailure == initialState;
  rebindPolicy == rebind;
  numOfFailure <= 10 failures/year;
  operationSemantics == atMostOnce;
};

SEProfile for ServiceI = profile {
  require ServiceExecutorReliability;
  require Reliability contract
    { dataPolicy == invalid; };
};
```

FIG. 24. Contract and binding for service

services can be executed when the system is up, which means that all resources must be available and consequently satisfy the high-availability requirements.

The reliability contract for the `Trader` (Figure 26) is based on a general contract (`HAServiceReliability`) for highly-available services. The contract is abstract in the sense that it only states the availability requirements and leaves several of the other dimensions unspecified. The `Trader` profile refines it by stating that the recovery time should be short.

In addition, we state that offer identifiers and object references returned by the trader are valid even after a failure. This means that an offer identifier returned before a failure can be used to withdraw an offer after the `Trader` has recovered. Also, any references returned by the `Trader` are valid during the `Trader`'s down period as well as after it has recovered, assuming, of course, that the services referred to by the references have not failed.

The start-up time for a service execution is very important; the time between a call is answered and the service starts executing must be short and definitely not more than one second. A start-up time that exceeds one second can make users believe there is a problem with the connection and therefore hang-up the phone, the consequence being both an unsatisfied customer and a lost business opportunity.

Having analyzed and estimated the execution times in the start-up execution path, we require that the `find` and `findAll` operations on the `Trader` respond quickly. We do not anticipate the throughput to constitute a bottleneck in this case.

We can relax the performance requirements for the `offer` and `withdraw` operations on the `Trader`. The reason being that these operations are not time critical from the service execution point of view. We specify the performance in Figure 26 as part of the `TraderProfile_P` profile.

The performance profile makes it clear that the implementation of `TraderI` should give invocations of `find` and `findAll` higher priority than invocations of `offer` and `withdraw`.

A resource service represents a pool of hardware and software resources that are expected to be highly-available. If a resource service is down, it is likely that there are major hardware or software problems that will take a long time to repair. Since failing resource services are expected to have long recovery times, they need to have, in principle, infinite `MTTF` to satisfy high availabil-

```
ResourceReliability = Reliability contract {
  availability >= 0.99999;
  failureMasking ==  { failure };
  serverFailure == initialState;
  rebindPolicy == rebind;
};

PlayerReliability =
  ResourceReliability refined by {
    MTTR = 7200 sec;
    TTF   {
            percentile 100 > 2000 days;
            percentile 80 > 6000 days;
            mean >= 7000 days;
           };

    availability >= 0.99999;
    contAvailability >= 0.999999;
    failureMasking == failure;
    serverFailure == initialState;
    rebindPolicy == rebind;
    numOfFailure <= 0.1 failures/year;
    operationSemantics == least_once;
    dataPolicy == no_guarantees;
};

PlayerProfile_P for PlayerI = profile {
  require PlayerReliability;
};
```

FIG. 25. Contract and binding for resources

```
HAServiceReliability = Reliability contract {
  availability >= 0.99999;
  failureMasking == { omission };
  serverFailure == initialState;
  rebindPolicy == rebind;
  numOfFailure <= 10 failures/year;
  operationSemantics == once;
};


TraderProfile_P for TraderI = profile {
  require HAServiceReliability refined by {
     MTTR {
            percentile 100 < 60 ;
            variance <= 0.1;
          }
  };

  from offer.OfferId, result of find, findAll
    require Reliability contract
      { dataPolicy == valid; };

  from find, findAll require Performance
    contract { latency { percentile 90 < 50 }; };

  from offer, withdraw require Performance
    contract { latency { percentile 80 < 2000 }; };
};
```

FIG. 26. Contract and binding for the Trader

ity requirements. This does not mean that individual resource cannot fail, but it does mean that there must be sufficient redundancy to mask failures.

In Figure 25 we define a general contract, called `ResourceReliability`, for `ResourceI`. The contract captures that resources need to be highly available. Each specific resource type—such as `PlayerReliability`—will then refine this general contract to specify its individual QoS properties.

6.3 Discussion

The specification of reliability and performance contracts, and the analysis of inter-component QoS dependencies, have given us many insights and important guidance. As an example, it has helped us realize that the `Trader` needs to support fast fail-over and use a reliable storage. We also found that the reliability of resources is essential, and that, in this example system, resource services should be responsible for their own reliability. The explicit specification also allows us to assign well-defined values to various dimension which make design goals and requirements mreo clear.

QML allows detailed descriptions of the QoS associated with operations, attributes, and operation parameters of interfaces. This level of detail is essential to clearly specify and divide the responsibilities among client and service implementations. The refinement mechanism is also essential. Refinement allows us to form hierarchies of contracts and profiles, which allows us to capture QoS requirements at various levels of abstraction.

Due to the limited space of this paper, we have not been able to include a full analysis or specification of the example system. In a real design, we also need to study what happens when various components fail, estimate the frequency of failures due to programming errors, etc. We also need to ensure that the QoS contracts provided by components actually allows the clients to satisfy requirements imposed on them. There are various modeling techniques available that are applicable to selected types of systems; see Reibman et al. [14] for an overview.

In our case, high availability requirements for `CallHandler` have resulted in strong demands on other services in the application. Another design alternative would be to demand that components such as the `ServiceExecutor` can handle failing resources and switch to other resources when needed. This would require more from the `ServiceExecutor`, but allow resource services to be less reliable.

Despite the limitations of our example, we believe that it demonstrates three important points: QoS should be considered during the design of distributed systems; QoS requires appropriate language support; QML is useful as a QoS specification language.

Firstly, we want to stress that considering QoS during design is both useful and necessary. It will directly impact the design and make developers aware of non-functional requirements.

Secondly, QoS cannot be effectively considered without appropriate language support. We need a language that helps designer capture QoS requirements and associate these with interfaces at a detailed level. We also need to make QoS requirements and offers first class citizens from a design language point of view.

Finally, we believe the example shows that QML is suitable to support designers in involving QoS considerations in the design phase.

7. Related Work

Common object-oriented analysis and design languages, such as UML [2], Objectory [13], Booch notation [1], and OMT [11], generally lack concepts and constructs for QoS specification. In some cases, they have limited support to deal with temporal aspects or call semantics [1].

Interface definition languages, such as OMG IDL [17], specify functional properties and lack any notion of QoS. TINA ODL [19] allows the programmer to associate QoS requirements with streams and operations. A major difference between TINA ODL and our approach is that they syntactically include QoS requirements within interface definitions. Thus, in TINA ODL, one cannot associate different QoS properties with different implementations of the same functional interface. Moreover, TINA ODL does not support refinement of QoS specifications, which is an essential concept in an object-oriented setting.

There are a number of languages that support QoS specification within a single QoS category. The SDL language [22] has been extended to include specification of temporal aspects. The RTSynchronizer programming construct allows modular specification of real-time properties [15]. These languages are all tied to one particular QoS category. In contrast, QML is general purpose; QoS categories are user-defined types in QML, and can be used to specify QoS properties within arbitrary categories.

Zinky et al. [20, 21] present a general framework, called QuO, to implement QoS-enabled distributed object systems. The notion of a *connection* between a client and a server is a fundamental concept in their framework. A connection is essentially a QoS-aware communication channel; the expected and measured QoS behaviors of a connection are characterized through a number of *QoS regions*. A region is a predicate over measurable connection quantities, such as latency and throughput. When a connection is established, the client and server agree

upon a specific region; this region captures the expected QoS behavior of the connection. After connection establishment, the actual QoS level is continuously monitored, and if the measured QoS level is no longer within the expected region, the client is notified through an upcall. The client and server can then adapt to the current environment and re-negotiate a new expected region.

QuO does not provide anything corresponding to refinement, conformance, or fine-grained characterizations provided by QML.

Within the Object Management Group (OMG) there is an ongoing effort to specify what is required to extend CORBA [17] to support QoS-enabled applications. The current status of the OMG QoS effort is described in [18], which presents a set of questions on QoS specification and interfaces. We believe that our approach provides an effective answer to some of these questions.

8. Discussion

Developing a QoS specification language is only the first step towards supporting QoS considerations in general and, as this paper suggest, as an integral part of the design process. We need methods that address the process aspects of designing with QoS in mind. For example, we need methods that help the designer make QoS-based trade-offs, and methods that help the designer decompose the application-level QoS requirements into QoS properties for individual components. In addition to methods, we also need tools that can check consistency and satisfaction of QoS specifications. For example, it would be desirable, to have a tool that can check whether a running service meets its QoS specification. Although a specification language is not a complete solution, we still believe it is an important step.

Specifying QoS properties at design time is only the starting point; eventually we need to implement the design and ensure that the QoS requirements are satisfied in the implementation. An important issue that must be addressed in the implementation, is what action to take at runtime if the QoS requirements cannot be satisfied in the current execution environment, for example, what should happen if the actual response time is higher than the stated response time requirement. In most applications, it is not acceptable for a service to stop executing because its QoS requirements cannot be satisfied. Instead, one would expect the service to adapt to its environment through graceful degradation.

For a service to adapt to its environment, it must be notified about divergence from specified requirements, and it must be able to dynamically specify relaxed requirements to the infrastructure, and to the services it depends upon, to communicate how it can gracefully degrade and thereby adapt to the current execution environment. We believe that our concepts of profile and contract can be used to specify QoS requirements at runtime as well as at design time. To facilitate runtime specification, we need profiles and contracts to be first class values in the implementation language. To achieve this, we can define a mapping from QML into the implementation language; for example, if the implementation language is C++, one could map contract types into classes and contracts into objects instantiated from those classes. The important thing to notice is that the *concepts* remain the same.

9. Concluding Remarks

In this paper we argue that taking QoS into account during the design of distributed object systems significantly influences design and implementation decisions. Late consideration of QoS aspects will often lead to increased development and maintenance costs as well as systems that fail to meet user expectations.

We have proposed a language, called QML, that will allow developers to explicitly deal with QoS as they specify interfaces. In this paper we show how QML can be used for QoS specification in class model and interface designs of distributed object systems. QML allows QoS specifications to be separated from interfaces but associated with uses and implementations of services. We propose a refinement mechanism that allows reuse and customization of QoS contracts. This refinement mechanism also allows us to deal with the interaction between QoS specification and interface inheritance; thus we truly support object-oriented design. We have also described how we can determine whether one specification satisfies and other with conformance checking. Finally, QML allows QoS specification at a fine-grained level—operation arguments and return values—that we believe is necessary in many applications and for many QoS dimensions.

Although this paper focused on the usage of QML in the context of software design, we intend to use it for the management of QoS in general. As an example, based on defined contracts and profiles, we intend to emit programming language definitions that can be used to construct concrete QoS parameters. Such parameters are used to offer and require QoS characteristics at the application programming interface level.

Our experience suggests that the concepts and language proposed in this paper will provide a sound foundation for future QoS specification languages and integration of such languages with general object-oriented specification and design languages.

References

1. Grady Booch. Object-Oriented Analysis and Design. Benjamin-Cummings Corp. 1994.
2. Grady Booch, Ivar Jacobson, and Jim Rumbaugh. Unified Modeling Language. *Rational Software Corporation*, version 1.0, January 1997.

3. Kenneth P. Birman. ISIS: A System for Fault-Tolerant Distributed Computing. Department of Computer Science, Cornell University. TR86-744, April 1986.

4. D. Coleman, P. Arnold, S. Bodoff, C. Dolin, F. Hayes, P. Jeremaes. Object-Oriented Development: The Fusion Method. Prentice-Hall. 1994.

5. Flaviu Cristian. Understanding Fault-Tolerant Distributed Systems. *Communications of the CACM*, Vol. 34, No. 2, February 1991.

6. Svend Frølund and Jari Koistinen. QML: A Language for Quality-of-Service Specification. Hewlett-Packard Laboratories, Tech. Report HPL-98-10, February, 1998.

7. Gray and Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann. 1993.

8. Jari Koistinen. Dimensions for Reliability Contracts in Distributed Object Systems. Hewlett-Packard Laboratories, Tech. Report HPL-97-119, October, 1997.

9. Bev Littlewood. Software Reliability Modelling. *In Software Engineer's Reference book*. Section 31, Butterworth-Heinemann Ltd.,1991.

10. Silvano Maffeis. Adding Group Communication and Fault-Tolerance to CORBA. Proceedings of USENIX Conference on Object-Oriented Technologies. June 1995.

11. Jim Rumbaugh et al. Object Modeling Language. Prentice-Hall, 1991.

12. J. H. Saltzer, D. P. Reed, and D. D. Clark. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems*, Vol. 2, No. 4, November 1984.

13. Ivar Jacobson, Magnus Christerson and Persson. Object-Oriented Software Engineering. Addison-Wesley, 1992.

14. A. L. Reibman and M. Veeraraghavan. Reliability Modeling: An Overview for System Designers. *IEEE Computer*, April, 1991.

15. Shangping Ren and Gul A. Agha. RTsynchronizer: Language Support for Real-Time Specifications in Distributed Systems. *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*, La Jolla, California, June 1995.

16. Richard Staehlin, Jonathan Walpole, and David Maier. A quality-of-service specification of multimedia presentations. Multimedia Systems, 3, 1995.

17. Object Management Group. *The Common Object Request Broker: architecture and specification*, July 1995. revision 2.0.

18. Object Management Group. *Quality of Service: OMG Green paper*. Draft revision 0.4a, June 12, 1997.

19. TINA Object Definition Language. Telecommunications Information Networking Consortium, June 1995.

20. John A. Zinky, David E. Bakken, and Richard D Schantz. Architectural Support for Quality of Service for CORBA objects. *Theory and Practice of Object Systems*, Vol. 3(1), 1997.

21. John A. Zinky, David E. Bakken, and Richard D Schantz. Overview of Quality of Service for Distributed Objects. *Proceedings of the Fifth IEEE conference on Dual Use*, May, 1995.

22. S. Leue. Specifying Real-Time Requirements for SDL specifications — a temporal logic-based approach. Protocol Specification, Testing, and Verification XV. Proceedings of the Fifteenth IFIP WG6. June, 1995.

Efficient Implementations of Java Remote Method Invocation (RMI) *

Vijaykumar Krishnaswamy Dan Walther Sumeer Bhola
Ethendranath Bommaiah George Riley Brad Topol

Mustaque Ahamad†

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Abstract

Java and the Remote Method Invocation (RMI) mechanism supported by it make it easy to build distributed applications and services in a heterogeneous environment. When the applications are interactive and require low response time, efficient implementations of RMI are needed. We explore both transport level protocols as well as object caching in the RMI framework to meet the performance requirements of interactive applications. We have developed a prototype system that offers new transport protocols and allows objects to be cached at client nodes. We describe the design issues and the implementation choices made in the prototype along with some preliminary performance results.

1 Introduction

Interactive applications that enable widely distributed users to cooperate over the Internet will become increasingly common in the future. Such applications have traditionally been explored in the area of groupware but as increased bandwidths become available into the home (e.g., cable and digital subscriber line (xDSL) networks), electronic commerce and entertainment applications will become interactive. For example, a consumer located at home can utilize a graphical user interface (GUI) to view various retail items he or she is interested in purchasing. Simultaneously, a sales associate located at the retail outlet may also have a copy of the GUI which permits the associate to see what the customer is selecting and may suggest alternatives which are then presented in the customer's GUI. Furthermore, the home consumer may have requested friends located at other homes to also participate in this decision making and therefore they too may be running a GUI and viewing the possibilities and also making suggestions. Many such interactive application scenarios can be constructed easily.

Interactive applications will be supported by shared distributed services. In order for the internetworked computing infrastructure to support the above application scenario, system support is needed to allow the distributed services and client applications to be programmed easily. The use of object technology is becoming an increasingly popular approach for implementing distributed services. This is due to the fact that object technology provides a uniform mechanism for accessing local and remote resources and reduces the complexity of building applications in an internetworked computing environment. The Java language is a popular foundation for building distributed services and applications because it hides the problems that

*This work was supported in part by NSF grants CDA-9501637 and CCR-9619371, and industrial partners of the Broadband Telecom Center, Georgia Institute of Technology.

†Contact author: Email and home page – mustaq@cc.gatech.edu, http://www.cc.gatech.edu/fac/Mustaque.Ahamad/.

arise due to heterogeneity of server and client hardware and software platforms. Remote Method Invocation (RMI) is Java's mechanism for supporting distributed object based computing [18]. RMI allows client/server based distributed applications to be developed easily because a client application running in a Java virtual machine at one node can invoke objects implemented by a remote Java virtual machine (e.g., a remote service) the same way as local objects.

Although RMI enhances the ease of programming for distributed applications, we have found that it does result in significant performance penalties for applications compared to message passing [10]. Such loss of performance is undesirable for interactive applications in a wide-area environment because of the need for interactive response time in the presence of high communication latencies. The additional processing required by RMI will add some overhead compared to message passing but there are a number of techniques that can exploit the communication structure embodied by RMI to provide better performance. For example, it may be possible to exploit the "invocation-response" nature of RMI communications to develop a more efficient communication protocol than the TCP protocol that is employed by RMI (such an approach was used in the implementation of remote procedure call or RPC [2] which is closely related to RMI). Furthermore, when possible, a client may be able to cache the state of remote objects and invoke them locally. In this case, the overhead associated with communication can be avoided when there is significant locality of access.

We explore a number of techniques to improve RMI performance and integrate them into the RMI framework. Since the performance of RMI depends on the underlying communication protocols, we first explore a number of alternate transports that may improve the performance of RMI implementations. We developed a user datagram protocol (UDP) based reliable message delivery protocol that exploits the request-response nature of RMI communications. Also, when object state is cached at client nodes, consistency of the replicated object copies has to be maintained. Consistency protocols for replicated objects can benefit from one-to-many (e.g., multicast) communication and we have developed a flexible multicast transport that is available to RMI implementation. Finally, we extend the reference layer in the RMI framework to cache objects at client nodes. This approach allows clients to transparently invoke remote objects independent of whether they are being cached. When a cached copy of an invoked object is available, the invocation is executed locally. An invalidation based protocol has been implemented to maintain consistency of the cached copies. All this support has been added to the RMI framework by extending interfaces that are provided in the framework. The prototype system we have implemented has allowed us to quantify the benefits of caching.

We briefly review the RMI framework in Section 2. This framework primarily consists of the transport layer and the reference layer. The transport layer provides interfaces for communication protocols that support message passing across sites. Section 3 describes the new protocols that have been added by us to the transport layer. These include a UDP based reliable message delivery protocol and a multicast protocol that is used in maintaining the consistency of cached object copies. We explore design issues for object caching in the RMI reference layer and discuss our implementation is Section 4. Performance studies and their discussion is presented in Sections 5 and 6. We describe related work and conclude the paper in Sections 7 and 8.

2 The Java RMI Framework

The RMI framework [8] in Java allows distributed application components to communicate via remote object invocations. In particular, a client running at one node can access a remote service by invoking a method of the object that implements the service. Thus, the RMI framework enables applications to exploit distributed object technology rather than low level message passing (e.g., sockets) to meet their communication needs. A high level architecture of the RMI framework is shown in Figure 1.

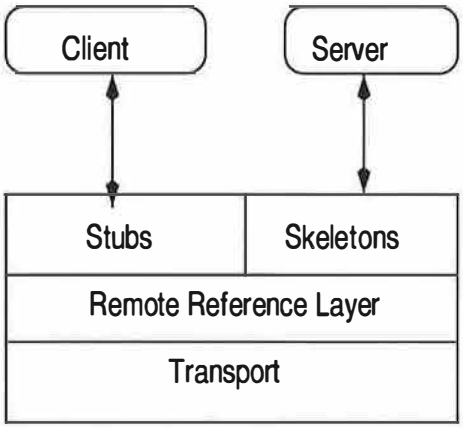


Figure 1: RMI Framework

All objects that can be invoked remotely must implement the interface **Remote**. This interface is just a tag that is used to distinguish remote objects from normal objects. Remote objects implement one or more interfaces and only through these interfaces are they visible to the outside world. The **rmic** tool is used to generate the skeleton and stub classes for a given remote object interface. For a given remote object **impl**, the stub, **impl_Stub**, and skeleton, **impl_Skel**, have the same set of methods that are defined in the interface of **impl**. **impl** provides the actual implementations of the methods defined in its interface.

A server that wants to make an object implemented by it remotely invokable must first *export* the object. This results in the instantiation of the skeleton object, the stub object, the reference object and the transport endpoint object in the server's virtual machine. When this object is bound onto a name server (e.g., **rmiregistry**) via the *bind* or the *rebind* operation, the stub, the client side reference object and the transport object are serialized and moved into the name server. At this point, the object is available for remote invocations from client nodes. To invoke a remote object **impl**, a client must first obtain a reference for it. Such a reference can be obtained in one of two ways. A client can do a *lookup* on the object which results in the instantiation of **impl_Stub** and other needed reference layer objects in the client's VM. All subsequent method invocations by the client to the remote object are routed via these objects. The client can also receive a reference to the object as an argument of an invocation. At the time the object reference is unmarshaled, the **impl_Stub** and other related objects are instantiated to enable the client to remotely invoke **impl**.

A client making an invocation on a remote object actually makes the call to the **stub** object. The *remote reference layer* is responsible for carrying out the invocation. The *transport layer* is responsible for connection setup, connection management and keeping track of and dispatching to remote objects. The **skeleton** for a remote object makes an upcall to the remote object implementation when a request for remote invocation is received at the server VM. Once the invocation is executed, the return value is sent back to the client via the skeleton, remote reference layer and transport layer on the server side, and then up through the transport and remote reference layers, and stub object on the client side. If an exception is thrown while making a call on the server side, this exception object rather than the result is marshaled and sent back to the client. The client side has enough machinery to detect that the received result is actually an exception rather than the result of the call and throws the corresponding exception to the application. A distributed garbage collector running in the server VM keeps track of client references for the remote object **impl**. In particular, client nodes *lease* **impl** for a certain period of time and each client reference increments its reference count by one. If a client's lease ends, the reference gets decremented and when this reference count becomes zero, **impl** can be garbage collected.

3 Efficient Communication Support

Although the RMI transport layer is flexible enough to include several transport protocols, at the time we started this work, only the Transmission Control Protocol (TCP) was available for RMI related communica-

tion. The more efficient User Datagram Protocol (UDP) cannot directly be used since it does not guarantee reliable delivery of invocation request and response messages. We take an approach that implements a reliable message delivery protocol based on UDP. We call this protocol R-UDP. Since RMI communications follow a "request-response" pattern, it is possible to exploit this structure in R-UDP to efficiently implement the reliable delivery of messages. In addition to R-UDP, we provide a flexible multicast protocol that can be used by the RMI reference layer when object caching is employed. These two protocols are described in this section. The impact of these protocols on the performance of remote invocations is discussed in a later section.

3.1 R-UDP: UDP based Reliable Protocol

We believe the use of a protocol like TCP for all phases of the RMI communication activity leads to certain inefficiencies. In particular, during the actual remote object invocation phase (after a given object has been located on the remote host and all necessary initialization has been performed), the data flow would typically fall into a "request-response" model, with the client sending a single "request" to the server, followed always by the server sending a single "reply" back to the client. Given this, any explicit acknowledgments used by TCP for requests can be avoided in a reliable protocol that is aware of the structure of RMI communication. A similar argument was used in the implementation of remote procedure calls by Birrell and Nelson in [2]. Another area where we expect to gain some performance improvements is by having explicit control over the behavior of the transport layer, specifically in the buffering and sending of network packets, rather than allowing the underlying protocol to make decisions about when to buffer data and when it is time to send a network packet.

3.1.1 Implementation Details

For communication between a client and server running at different sites, RMI allows for the specification of a "Socket Factory" by both the client and server. Thus, the default classes `Java.net.Socket` and `Java.net.ServerSocket` do not need to be used. We designed and implemented `RMISocket` and `RMIServerSocket` classes as subclasses of `Socket` and `ServerSocket` respectively. Since they are subclassed from the standard TCP socket classes, they must mimic their functionality. These new classes will allow a client/server pair to choose the R-UDP protocol for reliable message delivery on a *setSocketFactory* call. All other socket related processing in the RMI implementation is unchanged. The implementation of R-UDP can be broken down into two activities, (1) connection setup, and (2) reliable sending and receiving of data.

Connection Setup: During the connection setup phase, the server side *accept* method is simply blocked on a receive on the specified well known port address. A client wishing to connect to the server creates a local socket bound to a transient port, assigns a random 64 bit sequence number, and forwards the sequence number and local port number to the server in a datagram marked as a "Connection Request". Upon receipt of the connection request packet, the server creates a local socket bound to a transient port, assigns its own random 64 bit sequence number, and returns the sequence number and local port number to the client in a datagram marked as a "Connection Acknowledgment". When the client receives this packet, the connection is established. Of course, the Connection Request/Connection Ack sequence must be timed out and retransmitted in the event of errors.

Reliable Data Transfer: When either the client or server sends data, the normal `Java.net.Socket` paradigm is used, namely the use of `getDataOutputStream` and the writing of stream data to the returned output stream. Our implementation returns an output stream object of our design, which is a subclass of `ByteArrayOutputStream`. Our stream simply places all data written into a byte array buffer until a call to *flush* is made on the stream. When the flush call is made, the array is passed to a separate thread (the "SendingThread") to be trans-

mitted to the peer. The sending thread is blocked on a Java *wait* call until something is available to be sent, and is started by a *notify* call by the flush method. The data to be sent is placed in a datagram along with the next sequence number, an implicit acknowledgment sequence number (discussed later), and the actual data. The datagram is sent to the peer, marked as "Data packet, no acknowledgment required". The sending thread then blocks on a *wait* until an implicit acknowledgment is received (discussed next) or a constant timeout period has elapsed. If the timeout period elapses without receipt of an implicit acknowledgment, the sending thread re-sends the packet, but the second (and subsequent) tries are marked as "Data packet, explicit ack requested". As previously mentioned, all data transmissions to a peer include an "implicit ack", which notifies the peer of the highest sequence number packet that has been received. This allows for a server "reply" packet to serve as the acknowledgment that a client "request" packet has been received. This works well in the "request-response" data transmission model used for remote object invocations.

A host wanting to receive data from a peer uses the normal paradigm of `getInputDataStream` and reading stream data from the returned object. Our implementation returns an input stream object of our creation, which is a subclass of `ByteArrayInputStream`, and which is managed by a separate "Receiving" thread. The receiving thread is blocked on a datagram receive call, and will fill data in the byte array based on the contents of the received datagram. If an explicit acknowledgment is requested by the peer, an acknowledgment packet is prepared and returned, otherwise the thread just blocks waiting for the next message.

In the interest of brevity, the above discussion glosses over or ignores completely many of the details of a good implementation for reliable data transmission, such as the recognition and processing of duplicate data blocks. By no means is our implementation a fully functional TCP implementation, but is adequate for our needs in testing remote objects.

3.2 Multicast Communication

The RMI design is flexible enough to add server replication for improved scalability and fault-tolerance. We also explore object caching at client nodes to avoid the network latency when there is locality of access. Consistency protocols need to be employed when multiple copies of objects exist either due to replication or caching. Such protocols can benefit from multicast communication. By using multicast as against multiple unicast channels, we stand to gain in terms of better usage of network and server resources. For example, if an invalidation protocol is used to maintain consistency of replicated object copies, it is clearly beneficial to deliver the invalidation request to all the clients using a multicast message. However, we note that the scalability attainable can be limited by the consistency protocols even when multicast is used. In the case of invalidation protocols, for example, if the protocol requires responses from every client caching the object, then the scalability levels attainable are limited (we are exploring other consistency protocols that do not suffer from this problem).

We have implemented a reliable multicast framework along the lines of SRM [6], with a few novel changes. Like SRM, we use application data unit framing, negative acknowledgments, and multicast the retransmit request and response messages to the whole group. However, while SRM aims at *eventually* delivering *all* messages sent to the group, we aim at *eventually* delivering only the *essential* messages to all the members of the group. This is motivated from the fact that the multicast facility is intended to be used primarily by consistency related messages. Thus, we can rely on hints from the consistency protocol in identifying the *essential* messages. For example, if successive multicast messages update the state of the cached objects, the consistency protocol might permit loss of earlier updates as long as newer updates are delivered, that is, a newer update makes an earlier update *inessential*. We do not expend resources towards reliably delivering messages that have been identified as *inessential*. We believe that we stand to benefit significantly

from the above relaxed definition of reliability if the fraction of messages identified as *inessential* is reasonably high.

As we mentioned above, a retransmit request for a missed packet is sent to the whole group, and every member which can service this request locally enqueues a response with a random timer associated with it. The response is eventually sent out by the member whose timer expires the earliest. One of the main disadvantages of this scheme is that every member is required to participate in servicing retransmission requests. We propose to provide an option to permit the usage of a separate group address for multicasting retransmission requests and responses [12].

We have implemented the multicast protocol and used it to send invalidation messages in the consistency protocol that has been implemented in the prototype. The performance improvements made possible by multicast communication are discussed in Section 5.

4 Object Caching in the RMI Framework

Caching of remote objects has been shown to lead to better performance in systems that range from file systems to distributed shared memories. Clearly, if there is locality of access, caching a remote object at the client site can improve application performance because methods invoked on the object can be executed locally. In the RMI framework, the reference layer, which comes between the stub/skeleton objects and the transport layer, is responsible for handling remote method invocations. Thus, the reference layer is the natural place for providing alternative implementations of remote method invocations (e.g., using caching). We first discuss the design issues related to object caching at the reference layer and then present implementation details of a prototype system that we have developed for object caching. Our design of caching in the RMI framework was motivated by the following requirements.

1. The decision on whether an object is cacheable or not should be decided by the object provider at runtime, and not at compile time. This allows a single implementation of the object's functionality to be easily reused in different scenarios. This is all the more important in Java, because only single inheritance is available.

2. Cacheable objects should coexist with uncacheable (UnicastRemoteObjects) objects. This means that cacheable objects can have references to uncacheable objects and vice versa.

3. Caching should be transparent to the client[1], i.e. a client should not treat cacheable and uncacheable objects differently. Also, the invocation, failure and garbage collection semantics should be as close to uncacheable objects as possible.

We first describe an abstract model of RMI, and then show how caching can be added to it.

4.1 Abstract RMI

Based on the discussion in Section 2, we have presented a model of a non-caching RMI in Figure 2(a). The server P_s, first creates the server object O. It then exports O using a certain reference layer which creates the other four objects at P_s: (1) C, the client stub, (2) S, the server skeleton, (3) Ref_c, the object that implements the functionality of the client side reference and transport layer, and (4) Ref_s, the object that implements the server side reference and transport layer functionality. Thus, we have combined the reference and transport layer functionality into a single object. The server then binds C to a name server, which results in the marshaling[2] of the C and Ref_c state and unmarshaling at a name server. A lookup request of a client will be sent to the name server which will

[1] There are obviously situations where a client doesn't want to cache, for example, due to local memory limitations. These policies can be expressed in a separate policy object at the client and do not need to be part of the main control flow.

[2] Referred to as serialization in Java.

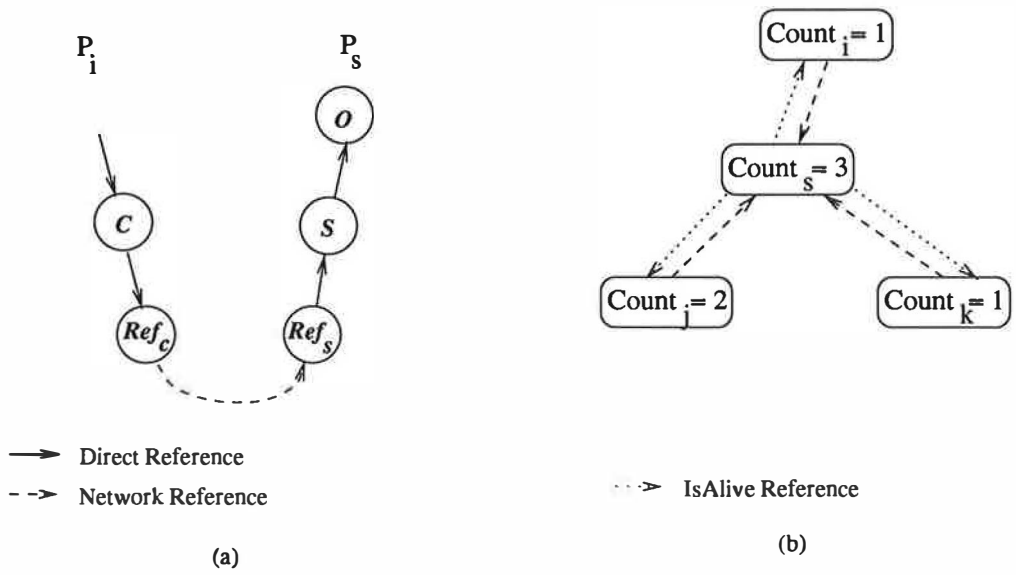


Figure 2: A Model of Non-caching RMI

send the C and Ref_c objects to the client (say P_i) in a similar manner. These objects and the references they hold across them are shown in Figure 2(a) after the lookup operation has been completed. Figure 2(b) shows the reference counting mechanism used for garbage collection. The count at the server, $count_s$, is the number of clients that have a network reference to object O. Each client (say P_i) also maintains a count of the number of references it has to O. P_i creates $count_i$ when it first gets a reference to O. At that time, it also informs the server, which increments $count_s$ and creates an *IsAlive* reference to P_i. If P_i creates more references to O (e.g., by cloning), it only increments $count_i$. When the value of $count_i$ drops to zero, the server is informed, which decrements $count_s$. O can be garbage collected when $count_s$ is zero and there are no local references to O. Note that the *IsAlive* reference is used by the server to detect that a client has crashed, so that $count_s$ can be decremented. This prevents garbage collection from being stalled because some client failed without informing the server.

4.2 Adding Caching

To add caching to this framework, we provide a different reference layer $Cref$, with $Cref_s$ being the server side, $Cref_c$ the client side, and $Cref'_s$ the client caching layer. The creation of the server object, export, binding and lookup are still the same (except for a different reference layer). Figure 3(a) shows the scenario after a lookup has been done at P_i. As there are cases when a process may have a remote reference but may never make an invocation on it (for example, a name server), we only initiate caching at the first invocation. Figure 3(b) shows the scenario after the first invocation. O' is the copy of O cached at P_i. The main differences between Ref_c and $Cref_c$ are

- Initiate caching on first invocation, instantiate cached object, and redirect reference to the cached copy.
- Send every invocation to the cached copy using the direct reference.
- When marshaled (for example, when passed as a parameter to some other remote invocation), do not marshal the direct reference subgraph. This limits

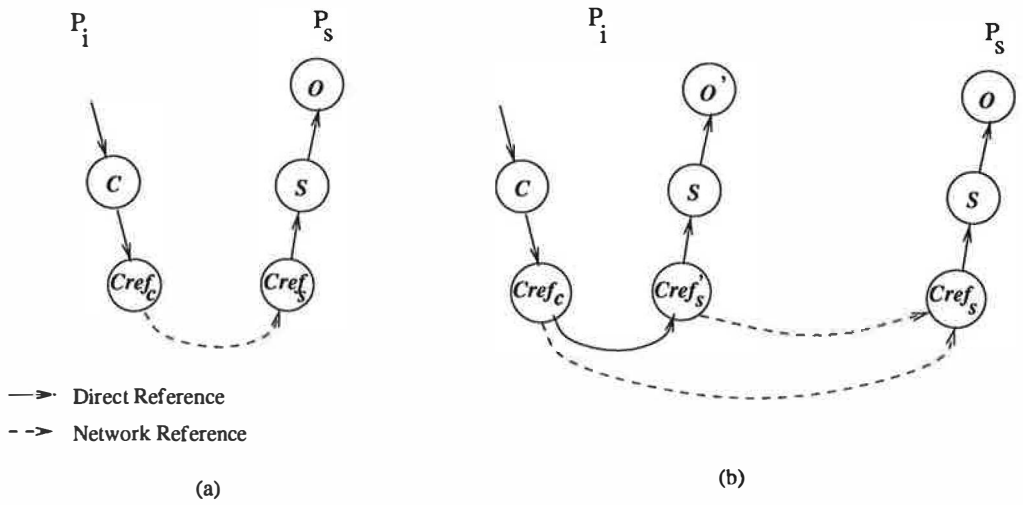


Figure 3: RMI with Caching

the architecture to a two level tree with $Cref_s$ at the root and $Cref'_s$s and $Cref_c$s as the leaves. We can also allow a policy where some $Cref_c$s never initiate caching i.e. their invocations always go to the $Cref_s$.

The job of maintaining the consistency of the cached object copy resides with $Cref_s$ and $Cref'_s$s.

Garbage Collection: A natural question to ask is how does caching effect RMI garbage collection. The answer is, it does not. As a $Cref'_s$ only exists in a virtual machine along with its $Cref_c$, we can still utilize the old system of counting network references from $Cref_c$s to $Cref_s$.

Specifying read and write methods: To initiate a consistency action, $Cref'_s$ needs to know whether a method invocation will only read the object state, or will also write it. We allow object programmers to provide information that can be used to infer if a method's execution only reads the object state or the state is also updated. In particular, the method code should include the throwing of read and write exceptions depending on how the state of the object is accessed by the method. Information about such exceptions is available from the class meta-data that is created by the Java compiler. We extended **rmic** to consult this meta-data to generate another object, `impl_MemFuncStat`, along with the `impl_stub` and `impl_skel` objects. `impl_MemFuncStat` implements a member function *isWriteMethod* that returns whether a certain method reads or modifies the state of `impl`.

Failure Semantics: We consider two types of failures, server, and client. In case of server failure, both noncaching RMI and our caching extension behave the same way, they stop working. Client failures in noncaching RMI don't lead to any problems (except for the GC mechanism detecting the failure and decrementing the counter). For caching clients in our caching RMI, the situation is somewhat complicated and depends on the cache consistency protocol. For an invalidation protocol, the crash of a client which did not have the only valid copy is easy to handle. The problem is when a client with the only valid copy crashes. A simple solution is for the server to use the last version it has as the valid copy and continue from there. This is perfectly acceptable if any updates done by the crashed client which were lost (and updates done by the same client after those lost updates), did not effect some part of the

world which can still be seen by the remaining clients. For example, assume that client P_1 updates a cached copy of object O_1 (whose server is at P_1), then updates remote object O_2 which resides at P_2, and then crashes. The update on O_2 is visible but the previous update on O_1 might have been lost because the new O_1 value was not yet sent to the server.

4.3 Implementation of Object Caching

The functionality provided by the $Cref_c$, $Cref_s$ and $Cref_s'$ objects shown in Figure 3 has to address a number of problems. First, to cache an object at a client site, the object state and implementation have to be made available to the client. The serialization interface provided by Java is used for transporting object state across nodes. If an object has state that is meaningful only at the server node where the object is instantiated (e.g., open network connections), new serialization methods are allowed by our system that override the default Java serialization methods. The implementation of an object is in the form of the bytecode which can be transferred to client nodes (stub bytecode is already transferred from server to client node in Java RMI). The client side is initially provided minimal code and whenever the system faults on the bytecode, a central code base specified by the server side is contacted and the necessary bytecode is downloaded on demand.

The execution of a method with a cached copy either only reads the object state or it also modifies the object state. The consistency protocol actions that need to be executed depend on whether the method will read or update the object state. We use the the `impl_MemFuncStat` object described earlier for object `impl` to determine the access type.

We employ the standard invalidation protocol to maintain consistency of cached object copies. Thus, when a client invokes a method of an object that can update the object state, the client communicates with the server node. The server keeps track of the clients that have copies of the object and sends them invalidation messages. Once copies at other clients are invalidated, the client that updates the object state is allowed to execute the method with the cached object copy.

We considered the following two approaches for designing a consistency framework that implements the invalidation protocol as well as other consistency protocols.

1. The implementation of a cacheable object extends a consistency object whereby it inherits all the methods of the consistency object which are invoked to maintain the object's consistency.

2. The caching framework maintains a reference for a consistency object and all invocations on the implementation get monitored by this consistency object.

The first approach means that the consistency protocols for a cached object are decided at compile time. The second approach not only allows us to dynamically link an object with a consistency protocol at runtime, it also allows for object caching to be enabled or disabled during the life of the object. Further, consistency levels and hence protocols can be changed depending on the degree of coupling required among the clients. Since the second approach provides more flexibility, we decided to use it in our implementation of object caching. All the consistency objects are derived from a base object called `ConsistencyModel` which has a generic set of methods that are common to all consistency objects. A particular consistency protocol (e.g., server initiated invalidations) is implemented by a specialized object that extends the ConsistencyModel object. Instantiation of this consistency object during the deserialization of the client side caching framework also forks a consistency thread which performs all the consistency actions on the object in a synchronized manner with the application thread using the object.

4.3.1 The Caching Framework

Some of the objects that make up the caching framework on the server and client sides are shown in Figures 4 and 5. On the server side, the reference object, `CacheableServerRef`, maintains state information so as to instantiate an object either in the caching or non-caching mode. It can also dynamically disable or enable caching. The reference layer objects, `CacheableServerRef` and `CacheableRef` are obtained by extending the default RMI reference classes `UnicastServerRef` and `UnicastRef`. When the remote object, `Impl` shown in Figure 4 is instantiated on the server, the object implementor provides enough information regarding the nature of caching, the transport protocol to be used, the consistency algorithm to be used etc. During instantiation, `Impl` calls the *exportObject* method of another class called `CacheableRemoteObject`. This instantiates `CacheableServerRef`, the reference layer object on the server side, `CacheableRef`, the reference layer object for the client, the consistency object (if needed, depending on the protocol), the transport endpoint object for the server VM and a few other objects. The configuration information specified by the user is stored in a `SystemParams` object which is also part of the reference layer. `SystemParams` object stores the name of the implementation, the name of the skeleton, and the *codebase* as its state. When a *bind* or a *rebind* operation is done on `Impl`, the `CacheableRef` object, the transport endpoint object and several other objects are serialized and exported to the **rmiregistry**. When a client does a *lookup* operation, `CacheableRef` and the transport endpoint object are transferred to the client's VM and are instantiated there. A `ConsistencyObject` object is instantiated as a part of the state of the `CacheableRef` object. During its instantiation, `Impl`, the skeleton for it and the `Impl_MemFuncStat` objects also get instantiated at the client. A consistency thread which handles the consistency requests from the server in a synchronized manner with the application thread is also forked during this process. The client side reference layer, from now on forwards all the invocations to the consistency object.

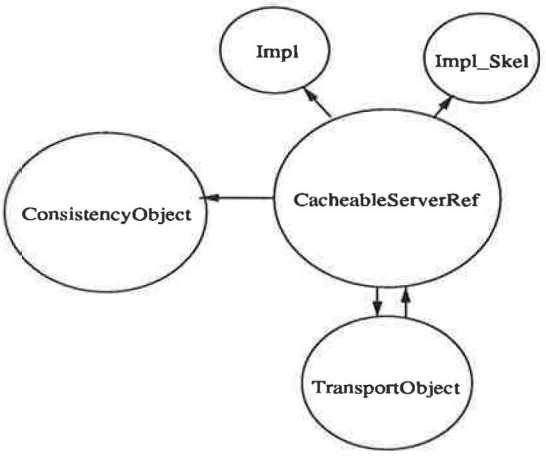


Figure 4: Server side object hierarchy for caching

The `ConsistencyObject` object performs the following sequence of actions before the actual invocation on the cached object is permitted.

- The cached copy is checked for its validity. If not valid then the most recent version of the copy is requested from the server (the server may communicate with another client to receive the latest copy). During this process, the client also acquires a *readlock* for the object that pins the object locally to ensure that its state cannot be invalidated while a method execution is in progress.
- The `impl_MemFuncStat` object is consulted to decide on the nature of the call. If it happens to be a write method, then the required consistency actions are executed and a *writelock* is acquired. This write lock provides atomicity of the method execution when the object state is updated.
- It then does the method invocation on the local object.

Finally when the application thread is about to quit, the inbuilt GarbageCollector is called in to free the resources. The consistency daemon's destruction method is modified so that if the client has the most recent

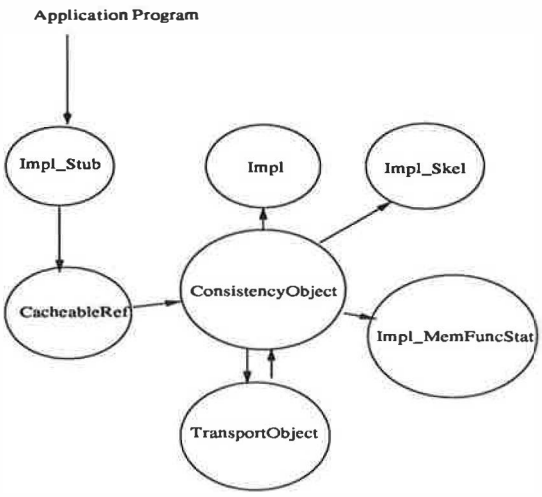


Figure 5: Client side object hierarchy for caching

copy of the object, it is sent to the server before the object is destroyed.

5 Performance Evaluation

We have discussed a number of techniques that could provide improved performance for RMI. The goal of this section is to experimentally evaluate the performance improvements (if any) that are made possible by the alternate transports and by object caching. Our performance studies are preliminary but the experiments conducted by us do provide some evidence of the effectiveness object caching and the use of multicast communication for maintaining consistency of cached object copies.

Our experiments were conducted in two different environments. In the first one, controlled experiments were conducted on a cluster of Sun Ultra 2's connected with a 100 Mbs Ethernet. There were no other applications running on the nodes in the cluster at the time of the experiments and as a result, we were able to reproduce the same results multiple times. We ran each experiment, for which results are presented in this section, six times. We present the average execution for remote method execution. The standard deviation across these experiments is not included because it was insignificant. For example, the maximum standard deviation observed in these experiments was 0.04.

Since caching is more effective when communication latencies are higher, the second environment we use in our experiments is two clusters connected via the Internet. These clusters were at the Georgia Tech and Emory University campuses which are separated by approximately six miles. The cluster at Emory had Sun Sparc 20 machines rather than Ultras. Although we could not control network traffic in the second environment which could impact the results of the experiments, we conducted the experiments late at night when there was minimal interference from other applications. As a result, we were able to obtain repeatable results with small variance across ten runs for most of the experiments. We present the mean times as well as the standard deviation for these experiments. In one case that required communication across several nodes in the wide-area environment, we could not obtain consistent results across different runs due to the variability in the environment. These results are not reported here. Thus, all the results reported here were obtained across a number of runs (at least six for each case) and we present both the mean and the standard deviation for them. In both environments, we used the JDK 1.1.5 distribution of Java with the just-in-time (JIT) compiling feature.

5.1 Object Caching

Object caching allows a remote invocation to be completed locally under a number of conditions. For example, if the valid state of the object is cached locally and the execution of an invocation only reads the object state, the needed consistency actions do not require remote communication. Similarly, if the node caches the object in exclusive mode and the object state is updated by the invocation's execution, other nodes are not notified of the update. If a copy of the object does not exist locally, communication with the server is required. When the object is requested in exclusive mode, the server may have to in-

validate other copies before it can return the state of the object to the requesting client node. To measure the costs of invocations under these different conditions, for different size arguments, we measured the costs of completing a remote invocation in the following cases.

- The object is invoked remotely at the server node without caching it locally. Thus, this is the base case where the RMI framework is used to execute the invocation remotely.
- The object is cacheable but at the time it is invoked, its valid state is not available at the client node. In this case, the client must request the current state of the object before the invocation can be executed locally.
- A valid copy of the object is in the cache and the invocation is executed locally. Furthermore, the object is cached in a mode such that the execution of the invocation does not result in communication with other nodes for maintaining consistency. This could be either because the execution of the invocation only reads the object state or in case of an update, the object is cached in an exclusive mode.
- The invocation is executed with a cached copy but communication with other nodes is necessary to maintain consistency. For example, if the state of the object is updated as a result of executing the invocation, read-only copies at other nodes must be invalidated. This is done by communicating with the server which sends invalidation messages to the other clients.

Table 1 shows the results of the experiments that were conducted to evaluate the effectiveness of caching in both cluster and wide-area environments. For the cluster environment, we present average invocation times of ten runs of each experiments. Since controlled experiments were done in the cluster, there was very low standard deviation across the runs (less than 0.04) and Table 1 does not show it for the cluster environment. Clearly, executing an invocation with a cached copy when no communication is required provides much better performance than invoking the object remotely. For example, in the cluster environment when the invocation argument size is 32 bytes, invocation with a cached object completed in 1.53 ms compared to 3.51 ms required when the invocation is executed remotely at the server. Since caching is transparent to the application invoking the object, the invocation arguments are marshaled before the point when the reference layer determines that the object is cached locally. As a result, the execution time in the caching case does include the marshaling and unmarshaling costs.

If the execution of a method with a cached copy does require consistency actions to be executed which result in communication with remote nodes (e.g., invalidation messages), then the execution time of an invocation with a cached copy degrades with the number of invalidation messages. In fact, if communication is required with the server or other clients, executing the invocation with a cached object copy requires more time than its execution at the server. For example, when a valid copy of the object is fetched from the server before locally executing the invocation, the invocation execution time with 32 byte argument size is 5.34 ms compared to 3.51 ms when the invocation is executed at the server. Thus, the benefits of caching to an application will depend on the locality of access and on the mix of invocations that read and update the state of the object. Caching will be effective only when after the caching of an object at a client node, the client executes a number of invocations locally. This will happen when access conflicts at different clients (e.g., object is updated at two clients or one reads it while another one writes the object) are rare.

We now consider the wide-area environment. In this environment, we present both average execution time and the standard deviation for ten runs of each experiment. The improvement in performance for an invocation that executes locally with a cached copy is more dramatic in the wide-area environment. The execution time for 32 byte argument size invocation with caching is 1.58

ms. which is almost 8 times faster than invoking the object at a remote server. Clearly, caching could be more effective when communication overheads are higher. Notice that the costs with cached objects are different in the cluster and wide-area environments. These differences are due to differences in the client hardware (Ultra Sparcs vs. Sparc20s). We were able to obtain consistent results across many runs of an experiment in the wide-area environment when either no messages were sent or messages were exchanged between only two nodes. In the case when the client executing the invocation had to communicate with the server, which in turn had to send an invalidation message to two other clients, we were not able to get consistent results across different runs of the experiments due to lack of control over the network environment. Thus, execution times for this case are not included in Table 1.

The results in Table 1 made use of the TCP transport to send all messages, including invalidation messages that are sent to maintain consistency of cached copies. Since an invalidation message has to be sent to multiple nodes, instead of using separate messages, the server can send a single multicast invalidation message to all nodes that need to invalidate their copies. We used the multicast transport developed by us to send invalidation messages. As shown in Table 2, the use of multicast does improve performance of object invocation when invalidation messages are sent to multiple nodes. For example, when copies need to be invalidated at four client nodes, the use of multicast reduces invocation execution time from 12.6 ms to 9.24 ms when the argument size is 32 bytes. Thus, it is desirable to include a multicast transport to support the communication required by consistency protocols when caching is employed.

The effectiveness of caching (e.g., overall performance improvement for an application when caching is employed) depends on the pattern of method invocations. Our measurements indicate that if there is locality of access (e.g., an object is accessed several times before it gets invalidated), caching can result in significantly better performance, especially when communication latencies are high. To precisely characterize the benefits of caching, actual application or workloads are necessary. We discuss this issue in the next section.

5.2 Reliable UDP Based Protocol

To evaluate the impact of a transport protocol on the performance of remote method invocation, we measured the cost of a remote invocation at the server node when TCP and R-UDP protocols are used as transports. We did these experiments in the cluster environment with various sizes of invocation arguments. These results are presented in Table 3. As can be seen, choosing the R-UDP transport does not provide better performance for remote method execution. In fact, for an invocation that has 32 byte size arguments, its execution at server with R-UDP takes 6.36 ms compared to 3.51 ms with TCP. Although we obtained better round trip message times (an invocation results in a request and a reply) for R-UDP at the transport level compared to TCP, R-UDP does not provide better execution times at the remote method invocation level. There are a number of reasons that can explain why invocation level performance is not improved by R-UDP.

We found that the assumptions made by R-UDP about the reference layer actually do not match what we observed. For example, R-UDP assumes that a `flush()` call is made when the reference layer wants an invocation request to be sent to the server and this is done only once for each invocation. We found multiple calls to `flush()`, including some when the stream had no data that needed to be sent. We fixed some of these problems but our use of several threads to manage the transmission, retransmission and acknowledgment of messages, and synchronization between these threads and the application thread resulted in significant overheads for R-UDP. Currently we are redesigning R-UDP to reduce some of these overheads.

Implementation of remote method execution	Invocation argument size in bytes	Invocation execution time in ms.		
		Cluster environment	Wide-area environment	
			Average	Standard Deviation
At server via RMI support	0	2.54	12.81	0.36
	32	3.51	13.61	0.24
	1024	3.95	15.24	0.49
At client with a valid cached copy, when no consistency related communication is needed	0	0.90	0.93	0.05
	32	1.53	1.58	0.04
	1024	1.61	1.65	0.04
At client, valid copy fetched from server, no other consistency related communication needed	0	4.70	25.89	0.38
	32	5.34	26.98	0.33
	1024	5.45	26.46	0.35
At client, valid copy of object available, two other client copies invalidated via server	0	8.51	-	-
	32	9.16	-	-
	1024	9.19	-	-

Table 1: Caching Performance

Communication generated for executing remote method	Invocation argument size in bytes	Invocation execution time in ms.	
		TCP protocol	Multicast protocol
Client communicates with server, server invalidates one other client	0	6.96	7.15
	32	7.54	7.60
	1024	7.64	7.87
Client communicates with server, server invalidates two other clients	0	8.51	7.28
	32	9.16	7.93
	1024	9.19	8.13
Client communicates with server, server invalidates four other clients	0	11.68	8.40
	32	12.60	9.24
	1024	12.68	9.38

Table 2: Caching Performance with TCP and Multicast Protocol

Invocation argument size in bytes	Invocation execution time in ms. in Cluster environment	
	TCP	R-UDP
0	2.54	5.33
32	3.51	6.36
1024	3.95	6.95

Table 3: RMI performance with different transports

6 Discussion

We have explored a number of techniques for developing efficient implementations of RMI and have integrated them in the RMI framework. The initial performance studies that have been done by us have helped us understand when these techniques may provide improved performance. Clearly, additional performance studies are necessary to quantify the improvements in RMI performance. First, there is lot of room for improving the performance of the new transports that have been added by us. These improvements could come from better thread management at the transport implementation level as well the use of just-in-time compiling to reduce overhead of user-level implementations of the new transports.

The performance benefits of object caching depend on the object access patterns at client nodes. In particular, the locality of access and read-write mix of object invocations play an important role in determining the effectivess of caching. We are exploring a range of interactive applications. In these applications, shared graphical user interfaces (GUIs) and visualizations at participating users are supported by several shared objects. To provide access time that is independent of network latencies, copies of such objects must be created at each participant site. Clearly, caching allows such copies to be made. Furthermore, the current *focus-of-attention* of the interactions only requires manipulations of a small number of the objects. Objects that are not part of the current focus-of-attention are not updated and their copies can be accessed locally to drive the shared GUIs. We feel that the periodic access required to refresh the shared GUIs and localized focus-of-attention would lead to access patterns that are desirable in a caching environment (e.g., most accesses will be read-only and cached copies will be accessed repeatedly). However, we have not implemented and evaluated the applications to quantify the benefits of caching. In our current and future work, we plan to explore several workloads and applications to evaluate the effectiveness of caching.

We were able to add the new transports and object caching by extending the interfaces provided by the RMI framework except a small number of modifications to the interfaces themselves. For example, we had to add a new method to the `RemoteProxy` class which returns the name of the stub for the given class, the `Remote` interface being implemented either by the class itself or by one of it's superclasses.

7 Related Work

We have explored a number of techniques for enhancing the performance of RMI. Communication protocols that exploit the request-response nature of communication in distributed applications include T-TCP [3], VMTP [4] and others. Reliable multicast communication has been studied extensively (Isis and related systems [1], SRM [6], RMTP [16], Log-based [7] and others). Our multicast protocol is designed specifically to meet the needs of object consistency protocols. As a result, it can offer optimizations that are not possible in generic protocols (e.g., messages with newer values of an object make messages containing overwritten values obsolete).

Object caching has been studied in systems such as Spring [15], Flex [11], Thor [14], Rover [9] and others. The Spring distributed operating system presented a generic architecture for object caching. There are several differences in the approaches taken by Spring and by us due to differences in the system environments. For example, separate cacher processes are employed by Spring because of the low overhead of inter-address space communication. Since such inter-address space communication support does not exist in Java, we chose to cache the objects in the virtual machine that invokes the objects. The Flex system that we had implemented previously focused on multiple consistency levels, and several caching design decisions made by it differed from object caching in Java. Also, Flex did not explore transport level support for fast remote invocations. Object replication and caching in Java independent of the RMI mechanism have been explored in sys-

tems such as TIE [5] and Mocha [17]. By incorporating caching in the RMI framework, we ensure that applications do not need to differently deal with cached and non-cached objects.

We chose a straightforward protocol for maintaining the consistency of cached objects (similar to one used in the Ivy system for maintainingg coherence of distributed shared memory pages [13]). Considerable work has been done in the area of object consistency and consistency protocols. For example, in distributed file systems and distributed shared memories, a number of protocols have been developed. In our future work, we will explore different consistency levels and consistency protocols by developing a consistency framework similar to the one developed in Flex [11].

8 Concluding Remarks

Interactive distributed applications programmed with Java can run on a wide range of platforms. However, the interactive response time needs of such applications in high communication latency environments require efficient support for communication across sites. We have explored efficient implementations of Java RMI because it allows distributed applications to interact via the remote object invocation mechanism. We were able to integrate a range of performance enhancing techniques in the RMI framework by extending the interfaces provided by RMI. The prototype system we implemented allowed us to evaluate the performance benefits made possible by object caching as well as by multicast communication.

In the future we will undertake detailed performance evaluation of the system using actual applications and workloads. In addition, we will explore fault-tolerance via server replication and other notions of object consistency and associated consistency protocols that provide better scalability.

Acknowledgements

We would like to thank Ken Arnold of Sun Microsystems, the shepherd for this paper, and other anonymous referees. The feedback provided by them has improved the paper greatly.

References

[1] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, 1991.

[2] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Trans. on Comp. Sys.*, February 1984.

[3] R. Braden "RFC1644: T/TCP - TCP Extensions for Transactions Functional Specification" July 1994

[4] D. R. Cheriton, VMTP: A Transport Protocol for the Next Generation of Communication Systems *Proc. SIGcomm, pp. 406-415, 1986.*

[5] Michael Condict, Dejan Milojicic, Franklin Reynolds and Don Bolinger. Towards a World-wide Civilization of Objects. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 25-32, Connemara, Ireland, September 1996.

[6] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *ACM SIGCOMM 95, August 1995, pp. 342-356.*

[7] H.W. Holbrook, S.K. Singhal, and D.R. Cheriton. Log-based Receiver-Reliable Multicast for Distributed Interactive Simulation. In *Proceedings of SIGCOMM '95 , Cambridge, MA, August,1995. ACM SIGCOMM.*

[8] Java Remote Method Invocation Documentation, http://java.sun.com/products/jdk/1.1/docs /guide/rmi/spec/.

[9] A. D. Joseph, A. F. de Lespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: A Toolkit for Mobile Information Access. In *Proc. of 15th SOSP*, 1995.

[10] R. Kordale, V. Krishnaswamy, S. Bhola, E. Bommaiah, G. Riley, B. Topol and M. Ahamad. Middleware Support for Scalable Services. In *Proc. IEEE Workshop on Community Networking*, October 1997.

[11] R. Kordale, M. Ahamad and M. Devarakonda. Object Caching in a CORBA Compliant System. Usenix Systems Journal, 1996.

[12] S. Kasera, J. Kurose, and D. Towsley. Scalable Reliable Multicast Using Multiple Multicast Groups. Proc. of 1997 ACM Sigmetrics, April 1997.

[13] K. Li, and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM trans. on Comp. Sys.*, Nov 1989.

[14] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber., U. Maheshwari and L. Shrira. Safe and Efficient Sharing of Persistent Objects in Thor. *ACM SIGMOD*, 1996.

[15] Michael N. Nelson, Graham Hamilton, and Yousef A. Khalidi. A Framework for Caching in an Object-Oriented System. Sun Microsystems Laboratories Technical Report SMLI-TR-93-19.

[16] S. Paul, K. K. Sabnani, J. C. Lin, S.Bhattacharyya. Reliable Multicast Transport Protocol. *IEEE Journal on Selected Areas in Communications, Vol. 15, No. 3,April 1997, Pages 407-421.*

[17] B. Topol, M. Ahamad and J. Stasko. Robust State Sharing for Wide Area Distributed Applications, In *Proc. of International Conference on Distributed Computing (ICDCS)*, May 1998.

[18] Ann Wollrath, Roger Riggs , Jim Waldo. A Distributed Object Model for Java System. *Proceedings of the USENIX COOTS 1996.*

The Design and Performance of MedJava

A Distributed Electronic Medical Imaging System

Developed with Java Applets and Web Tools

Prashant Jain, Seth Widoff, and Douglas C. Schmidt
{pjain,sbw1,schmidt}@cs.wustl.edu
Department of Computer Science
Washington University
*St. Louis, MO 63130, (314) 935-4215**

Abstract

The Java programming language has gained substantial popularity in the past two years. Java's networking features, along with the growing number of Web browsers that execute Java applets, facilitate Internet programming. Despite the popularity of Java, however, there are many concerns about its efficiency. In particular, networking and computation performance are key concerns when considering the use of Java to develop performance-sensitive distributed applications.

This paper makes three contributions to the study of Java for performance-sensitive distributed applications. First, we describe an architecture using Java and the Web to develop MedJava, which is a distributed electronic medical imaging system with stringent networking and computation requirements. Second, we present benchmarks of MedJava image processing and compare the results to the performance of xv, which is an equivalent image processing application written in C. Finally, we present performance benchmarks using Java as a transport interface to exchange large medical images over high-speed ATM networks.

For computationally intensive algorithms, such as image filters, hand-optimized Java code, coupled with use of a JIT compiler, can sometimes compensate for the lack of compile-time optimization and yield performance commensurate with identical compiled C code. With rigorous compile-time optimizations employed, C compilers still tend to generate more efficient code. However, with the advent of highly optimizing Java compilers, it should be feasible to use Java for the performance-sensitive distributed applications where C and C++ are currently used.

*This research is supported in part by a grant from Siemens Medical Engineering, Erlangen, Germany.

1 Introduction

Medical imaging plays a key role in the development of a regulatory review process for radiologists and physicians [1]. The demand for electronic medical imaging systems (EMISs) that allow visualization and processing of medical images has increased significantly [2]. The advent of modalities, such as angiography, CT, MRI, nuclear medicine, and ultrasound, that acquire data digitally and the ability to digitize medical images from film has heightened the demand for EMISs.

The growing demand for EMISs has been coupled with a need to access medical images and other diagnostic information remotely across networks [3]. Connecting radiologists electronically with patients increases the availability of health care. In addition, it can facilitate the delivery of remote diagnostics and remote surgery [4].

As a result of these forces, there is also increasing demand for *distributed* EMISs. These systems supply health care providers with the capability to access medical images and related clinical studies across a network in order to analyze and diagnose patient records and exams. The need for distributed EMISs is also driven by economic factors. As independent health hospitals consolidate into integrated health care delivery systems [2], they will require distributed computer systems to unify their multiple and distinct image repositories.

Figure 1 shows the network topology of a distributed EMIS. In this environment, medical images are captured by modalities and transferred to appropriate Image Stores. Radiologists and physicians can then download these images to diagnostic workstations for viewing, image processing, and diagnosis. High-speed networks, such as ATM or Fast Ethernet, allow the transfer of images efficiently, reliably, and economically.

Image processing is a set of computational techniques for enhancing and analyzing images. Image processing techniques apply algorithms, called *image filters*, to manipulate

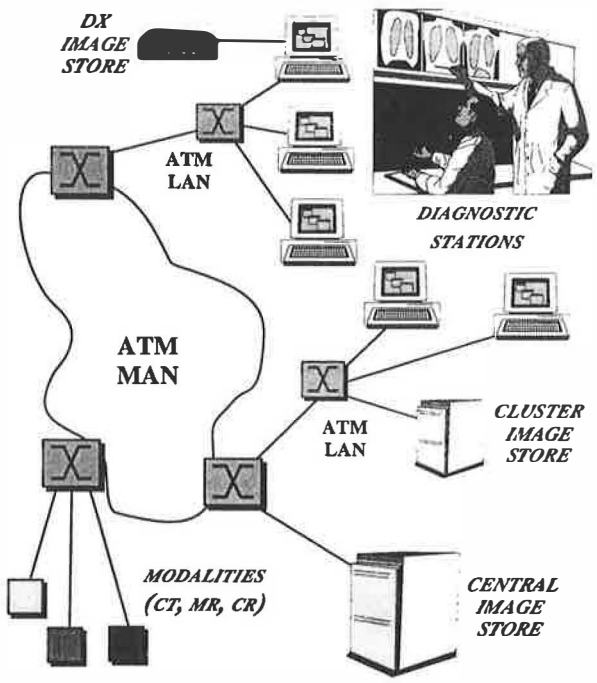


Figure 1: Topology of a Distributed EMIS

images. For example, radiologists may need to sharpen an image to properly diagnose a tumor. Similarly, to identify a kidney stone, a radiologists may need to zoom into an image while maintaining high resolution. Thus, an EMIS must provide powerful image processing capabilities, as well as efficient distributed image retrieval and storage mechanisms.

This paper describes the design and performance of *MedJava*, a distributed EMIS developed using the Java environment and the Web. The paper examines the feasibility of using Java to develop large-scale distributed medical imaging applications with demanding performance requirements for networking speed and image processing speed.

To evaluate Java's image processing performance, we conducted extensive benchmarking of MedJava and compared the results to the performance of `xv`, an equivalent image processing application written in C. To evaluate the performance of Java as a transport interface for exchanging large images over high-speed networks, we performed a series of network benchmarking tests over at 155 Mbps ATM switch and compared the results to the performance of C/C++ as a transport interface.

Our empirical measurements reveal that an imaging system implemented in C/C++ always out-performs an imaging system implemented using interpreted Java by 30 to 100 times. However, the performance of Java code using a "just-in-time" (JIT) compiler is ~1.5 to 5 times slower than the performance of compiled C/C++ code. Likewise, using Java as the transport interface performs 2% to 50% slower than using C/C++ as the transport interface. However, for sender buffer size close to the network MTU size, the performance of using Java as the transport interface was only 9% slower than the performance of using C/C++ as the transport interface. Therefore, we conclude that it is becoming feasible to use Java to develop large-scale distributed EMISs. Java is particularly relevant for wide-area environments, such as teleradiology, where conventional EMIS capabilities are too costly or unwieldy with existing development tools.

The remainder of this paper is organized as follows: Section 2 describes the object-oriented (OO) design and features of MedJava; Section 3 compares the performance of MedJava with an an equivalent image processing application written in C and compares the performance of a Java transport interface with the performance of a C/C++ transport interface; Section 4 describes related work; and Section 5 presents concluding remarks.

2 Design of the MedJava Framework

2.1 Problem: Resolving Distributed EMIS Development Forces

A distributed electronic medical imaging system (EMIS) must meet the following requirements:

- **Usable:** An EMIS must be usable to make it as convenient to practice radiology as conventional film-based technology.
- **Efficient:** An EMIS must be efficient to process and deliver medical images rapidly to radiologists.
- **Scalable:** An EMIS must be scalable to support the growing demands of large-scale integrated health care delivery systems [2].
- **Flexible:** An EMIS must be flexible to transfer different types of images and to dynamically reconfigure image processing features to cope with changing requirements.
- **Reliable:** An EMIS must be reliable to ensure that medical images are delivered correctly and are available when requested by users.
- **Secure:** An EMIS must be secure to ensure that confidential patient information is not compromised.
- **Cost-effective:** An EMIS must be cost-effective to minimize the overhead of accessing patient data across networks.

Developing a distributed EMIS that meets all of these requirements is challenging, particularly since certain features conflict with other features. For example, it is hard to develop an EMIS that is efficient, scalable, and cost-effective. This is because efficiency often requires high-performance computers and high-speed networks, thereby raising costs as the number of system users increases.

2.2 Solution: Java and the Web

Over the past two years, the Java programming language has sparked considerable interest among software developers. Its popularity stems from its flexibility, portability, and relative simplicity compared with other object-oriented programming languages [5].

The strong interest in the Java language has coincided with the ubiquity of inexpensive Web browsers. This has brought the Web technology to the desktop of many computer users, including radiologists and physicians.

A feature supported by Java that is particularly relevant to distributed EMISs is the *applet*. An applet is a Java class that can be downloaded from a Web server and run in a context application such as a Web browser or an applet viewer. The ability to download Java classes across a network can simplify the development and configuration of efficient and reliable distributed applications [6].

Once downloaded from a Web server, applets run as applications within the local machine's Java run-time environment, which is typically a Web browser. In theory, therefore, applets can be very efficient since they harness the power of the local machine on which they run, rather than requiring high latency RPC calls to remote servers [7].

The MedJava distributed EMIS was developed as a Java applet. Therefore, it exploits the functionality of front-ends offered by Web browsers. An increasing number of browsers (such as Internet Explorer and Netscape Navigator and Communicator) are Java-enabled and provide a run-time environment for Java applets. A Java-enabled browser provides a Java Virtual Machine (JVM), which is used to execute Java applets. MedJava leverages the convenience of Java to manipulate images and provides image processing capabilities to radiologists and physicians connected via the Web.

In our experience, developing a distributed EMIS in Java is relatively cost effective since Java is fairly simple to learn and use. In addition, Java provides standard packages that support GUI development, networking, and image processing. For example, the package `java.awt.image` contains reusable classes for managing and manipulating image data, including color models, cropping, color filtering, setting pixel values, and grabbing bitmaps [8].

Since Java is written to a virtual machine, an EMIS developer need only compile the Java source code to Java bytecode. The EMIS applet will execute on any platform that has a Java Virtual Machine implementation. Many Java bytecode compilers and interpreters are available on a variety of platforms. In principle, therefore, switching to new platforms or upgraded hardware on the same platform should not require changes to the software or even recompilation of the Java source. Consequently, an EMIS can be constructed on a network of heterogeneous machines and platforms with a single set of Java class files.

2.3 Caveat: Meeting EMIS Performance Requirements

Despite the software engineering benefits of developing a distributed EMIS in Java, there are serious concerns with its performance relative to languages like C and C++. Performance is a key requirement in a distributed EMIS since timely diagnosis of patient exams by radiologists can be life-critical. For instance, in an emergency room (ER), patient exams and medical images must be delivered rapidly to radiologists and ER physicians. In addition, an EMIS must allow radiologists to process and analyze medical images efficiently to make appropriate diagnoses.

Meeting the performance demands of a large-scale distributed EMIS requires the following support from the JVM. First, its image processing must be precise and efficient. Second, its networking mechanisms must download and upload large medical images rapidly. Assuming that efficient image processing algorithms are used, the performance of a Java applet depends largely on the efficiency of the hardware and the JVM implementation on which the applet is run.

The need for efficiency motivates the development of high-speed JIT compilers that translate Java bytecode into native code for the local machine the browser runs on. JIT compilers are "just-in-time" since they compile Java bytecode into native code on a per-method basis immediately before calling the methods. Several browsers, such as Netscape and Internet Explorer, provide JIT compilers as part of their JVM.

Although Java JIT compilers avoid the penalty of interpretation, previous studies [9] show that the cost of compilation can significantly interrupt the flow of execution. This performance degradation can cause Java code to run significantly slower than compiled C/C++ code. Section 3 quantifies the overhead of Java and C/C++ empirically.

2.4 Key Features of MedJava

MedJava has been developed as a Java applet. Therefore, it can run on any Java-enabled browser that supports the standard AWT windowing toolkit. MedJava allows users to download medical images across the network. Once an image has been downloaded, it can be processed by applying one or more image filters, which are based on algorithms in the C source code from `xv`. For example, a medical image can be sharpened by applying the Sharpen Filter. Sharpening a medical image enhances the details of the image, which is useful for radiologists who diagnose internal ailments.

Although MedJava is targeted for distributed EMIS requirements, it is a general-purpose imaging tool that can process both medical and non-medical images. Therefore, in addition

to providing medical filters like sharpening or unsharp masking, MedJava provides other non-medical image processing filters such as an Emboss filter, Oil Paint filter, and Edge Detect filter. These filters are useful for processing non-medical images. For example, edge detection serves as an important initial step in many computer vision processes because edges contain the bulk of the information within an image [10]. Once the edges of an image are detected, additional operations such as pseudo-coloring can be applied to the image.

Image filters can be dynamically configured and reconfigured into MedJava via the *Service Configurator* pattern [6]. This makes it convenient to enhance filter implementation or install new filters without restarting the MedJava applet. For example, a radiologist may find a sharpen filter that uses the unsharp mask algorithm to be more efficient than a sharpen filter that simply applies a convolution matrix to all the pixels. Doing this substitution in MedJava is straightforward and can be done without reloading the entire applet.

Once an image has been processed by applying the filter(s), it can be uploaded to the server where the applet was downloaded. HTTP server implementations, such as JAWS [11, 12] and Jigsaw, support file uploading and can be used by MedJava to upload images. In addition, the MedJava applet provides a hierarchical browser that allows users to traverse directories of images on remote servers. This makes it straightforward to find and select images across the network, making MedJava quite usable, as well as easy to learn.

To facilitate performance measurements, the MedJava applet can be configured to run in benchmark mode. When the applet runs in benchmark mode, it computes the time (in milliseconds) required to apply filters on downloaded images. The timer starts at the beginning of each image processing algorithm and stops immediately after the algorithm terminates.

2.5 The OO Design of MedJava

Figure 2 shows the architecture of the MedJava framework developed at Washington University to meet distributed EMIS requirements. The two primary components in the architecture include the MedJava client applet and JAWS, which is a high-performance HTTP server also developed at Washington University [12, 11]. The MedJava applet was implemented with components from Java ACE [13], the Blob Streaming framework [14], and standard Java packages such as `java.awt` and `java.awt.image`. Each of these components is outlined below.

2.5.1 MedJava Applet

The MedJava client applet contains the following components shown in Figure 2:

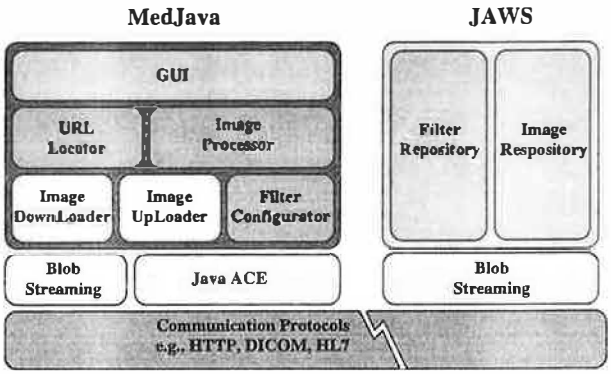


Figure 2: MedJava Framework

Graphical User Interface: which provides a front-end to the image processing tool. Figure 3 illustrates the graphical user interface (GUI) used to display a podiatry image. The

Figure 3: Processing a Medical Image in MedJava

MedJava GUI allows users to download images, apply image processing filters on them, and upload the images to a server.

URL Locator: which locates a URL that can reference an image or a directory. If the URL points to a directory, the contents of the directory are retrieved so users can browse them to obtain a list of images and subdirectories in that directory. The URL Locator is used by the Image Downloader and Image Uploader to download and upload images, respectively.

Image Downloader: which downloads an image located by the URL Locator and displays the image in the applet. The Image Downloader ensures that all pixels of the image are retrieved and displayed properly.

Image Processor: which processes the currently displayed image using the image filter selected by the user. Processing an image manipulates the pixel values of the image to create and display a new image.

Image Uploader: which uploads the currently displayed image to the server from where the applet was downloaded from.[1] The Image Uploader generates a GIF-format for the currently displayed image and writes the data to the server. This allows the user to save processed images persistently at the server.

Filter Configurator: which downloads image filters from the Server and configures them in the applet. The Filter Configurator uses the Service Configurator pattern [6] to dynamically configure the image filters.

2.5.2 JAWS

JAWS is a high-performance, multi-threaded, HTTP Web Server [11]. For the purposes of MedJava, JAWS stores the MedJava client applet, the image filter repository, and the images. The MedJava client applet uses the image filter repository to download specific image filters. Each image filter is a Java class that can be downloaded by MedJava. This design allows MedJava applets to be dynamically configured with image filters, thereby making image filter configuration highly flexible.

In addition, JAWS supports file uploading by implementing the HTTP PUT method. This allows the MedJava client applet to save processed images persistently at the server. JAWS implements other HTTP features (such as CGI bin and persistent connections) that are useful for developing Web-based systems.

Figure 4 illustrates the interaction of MedJava and JAWS. The *MedJava client applet* is downloaded into a Web browser from the *JAWS server*. Through *GUI* interactions, a radiologist instructs the MedJava client applet to retrieve images from JAWS (or other servers across the network). The *requester* is the active component of the browser running the MedJava applet that communicates over the *network*. It issues a request for the image to JAWS with the appropriate syntax of the *transfer protocol* (which is HTTP in this case). Incoming requests to the JAWS are received by the *dispatcher*, which is the request demultiplexing engine of the server. It is responsible for creating new threads. Each request is processed by a *handler*,

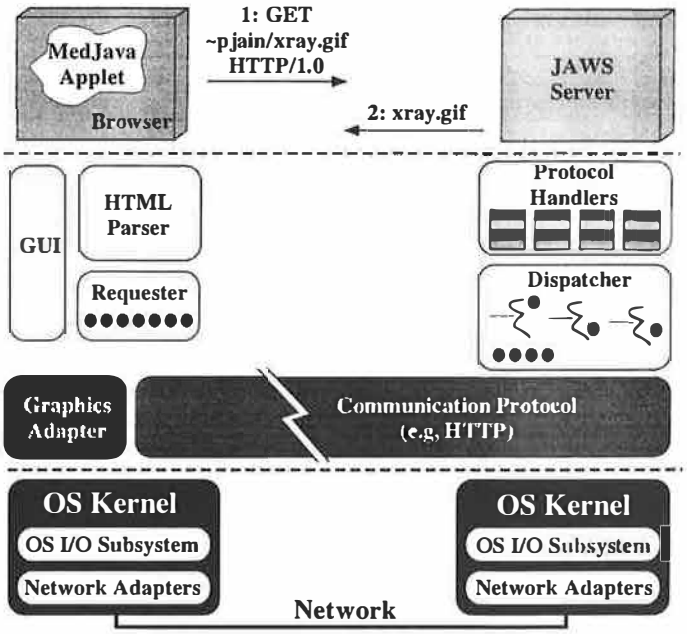


Figure 4: Architecture of MedJava and JAWS

which goes through a *lifecycle* of parsing the request, logging the request, fetching image status information, updating the cache, sending the image, and cleaning up after the request is done. When the response returns to the client with the requested image, it is parsed by an *HTML parser* so that the image may be rendered. At this stage, the *requester* may issue other requests on behalf of the client, *e.g.*, to maintain a client-side cache.

2.5.3 Blob Streaming

Figure 5 illustrates the Blob Streaming framework. The framework provides a uniform interface that allows EMIS application developers to transfer data across a network flexibly and efficiently. Blob Streaming uses the HTTP protocol for the data transfer.[2] Therefore, it can be used to communicate with high-performance Web servers (such as JAWS) to download images across the network. In addition, it can be used to communicate with Web servers that implement the HTTP PUT method to upload images from the browser to the server.

Although Blob Streaming supports both image downloading and image uploading across the network, its use within a Java applet is restricted due to applet security mechanisms. To prevent security breaches, Java imposes certain restrictions on applets. For example, a Java applet can not write to the local

[1] Due to applet security restrictions, images can only be uploaded to the server where the applet was downloaded from. In addition, the Web server must support file uploading by implementing the HTTP PUT method.

[2] Although the current Blob Streaming protocol is HTTP, other medical-specific communication protocols (such as DICOM and HL7) can also be supported.

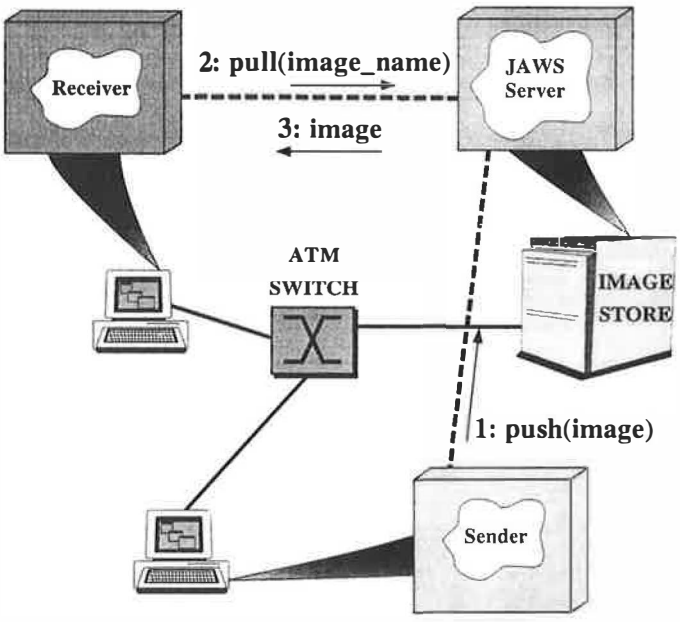


Figure 5: Blob Streaming Framework

file system of the local machine it is running on. Similarly, a Java applet can generally download files only from the server where the applet was downloaded. Likewise, a Java applet can only upload files to the server where the applet was downloaded.

Java applets provide an exception to these security restrictions, however. In particular, the Java `Applet` class provides a method that allows an applet to download images from any server reachable via a URL. Since the method is defined in the Java `Applet` class, it allows Java to ensure there are no security violations. MedJava uses this `Applet` method to download images across the network. Therefore, images to be processed can reside in a file system managed by the HTTP Server from where the MedJava client applet was downloaded or can reside on some other server in the network. However, Blob Streaming can only be used to upload images to the server where the MedJava applet was downloaded.

2.5.4 Java ACE

Java ACE [5] is a port of the C++ version of the ADAPTIVE Communication Environment (ACE) [15]. ACE is an OO network programming toolkit that provides reusable components for building distributed applications. Containing ~125,000 lines of code, the C++ version of ACE provides a rich set of reusable C++ wrappers and framework components that perform common communication software tasks portably across a range of OS platforms.

The Java version of ACE Contains ~10,000 lines of code, which is over 90% smaller than the C++ version. The reduction in size occurs largely because the JVM provides most of the OS-level wrappers necessary in C++ ACE. Despite the reduced size, Java ACE provides most of the functionality of the C++ version of ACE, such as event handler dispatching, dynamic (re)configuration of distributed services, and support for concurrent execution and synchronization. Java ACE implements several key design patterns for concurrent network programming, such as Acceptor and Connector [16] and Active Object [17]. This makes it easier to developing networking applications using Java ACE easier compared to programming directly with the lower-level Java APIs.

Figure 6 illustrates the architecture and key components in Java ACE. MedJava uses several components in Java ACE. For

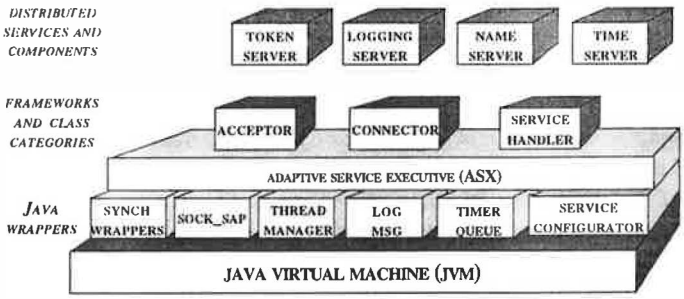


Figure 6: The Java ACE Framework

example, Java ACE provides an implementation of the Service Configurator pattern [6]. MedJava uses this pattern to dynamically configure and reconfigure image filters. Likewise, MedJava uses Java ACE profile timers to compute performance in benchmark mode.

3 Performance Benchmarks

This section presents the results of performance benchmarks conducted with the MedJava image processing system. We performed the following two sets of benchmarks:

1. Image processing performance: We measured the performance of MedJava to determine the overhead of using Java for image processing. We compared the performance of our MedJava applet with the performance of `xv`. `Xv` is a widely-used image processing application written in C. The MedJava image process applets are based on the `xv` algorithms.

2. High-speed networking performance: We measured the performance of using Java sockets over a high-speed ATM network to determine the overhead of using Java for transporting data. We compared the network performance results of Java to the results of similar tests using C/C++.

Below, we describe our benchmarking testbed environment, the benchmarks we performed, and the results we obtained.

3.1 MedJava Image Processing Benchmarks

We benchmarked MedJava to compare the performance of Java with xv, which is a widely-used image processing application written in C. Xv contains a broad range of image filters such as Blur, Sharpen, and Emboss. By applying a filter to an image in xv, and then applying an equivalent filter algorithm written in Java to the same image, we compared the performance of Java and C directly. In addition, we benchmarked the performance of different Web browsers running the MedJava applet.

3.1.1 Benchmarking Testbed Environment

Hardware Configuration: To study the performance of MedJava, we constructed a hardware and software testbed consisting of a Web server and two clients connected by Ethernet, as shown in Figure 7. The clients in our experiment were

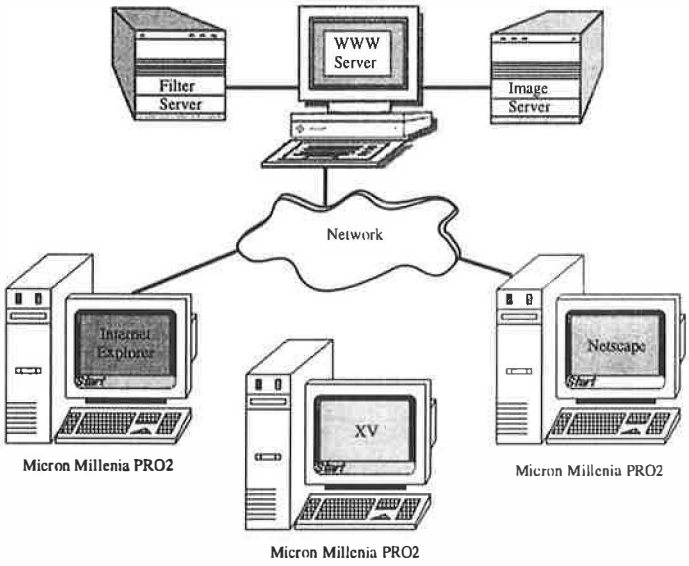


Figure 7: Web Browser Testbed Environment

Micron Millenia PRO2 plus workstations. Each PRO2 has 128 MB of RAM and is equipped with dual 180 Mhz PentiumPro processors.

JVM Software Configuration: We ran MedJava in two different Web browsers to determine how efficiently these browsers execute Java code. The browsers chosen for our tests were Internet Explorer 4.0 release 2 on Windows NT and Netscape 4.0 on NT. Internet Explorer 4.0 on NT and Netscape 4.0 on NT include Java JIT compilers, written by Microsoft and Symantec, respectively.

As shown in Section 3.1.4, JIT compilers have a substantial impact on performance. To compare the performance of the xv algorithms with their Java counterparts, we extracted the GIF loading and processing elements from the freely distributed xv source, removed all remnants of the X-Windows GUI, instrumented the algorithms with timer mechanisms in locations equivalent to the Java algorithms. We compiled this subset of xv using Microsoft Visual C++ version 5.0, with full optimization enabled.

Image filters can potentially require $O(n^2)$ time to execute. For large images, this processing can dominate the loading and display times. Therefore, the running time of the algorithms is an appropriate measure of the overall performance of an image processing application.

Image processing configuration: The standard Java image processing framework uses a "Pipes and Filters" pattern architecture [18]. Downstream sits an `java.image.ImageConsumer` that has registered with an upstream `java.image.ImageProducer` for pixel delivery. The `ImageProducer` invokes the `setPixels` method on the `ImageConsumer`, delivering portions of the image array until it completes by invoking the `ImageComplete` method.

The Pipes and Filters pattern architecture allows the `ImageConsumer` subclass to process the image as it receives the pieces or when the image source arrives in its entirety. An `ImageFilter` is a subclass of `ImageConsumer` situated between the producer and consumer who intercepts the flow of pixels, altering them in some way before it passes the image to the subsequent `ImageConsumer`. All `ImageFilters` in this experiment override the `ImageComplete` method and iterate over each pixel. The computational complexity for each filter depends on how much work the filter does during each iteration.

We selected the following seven filters, which exhibit different computational complexities. These filters are available in both xv and MedJava, and are ranked according to their usefulness in the domain of medical image processing.

1. Sharpen Filter: which computes for each pixel the mean of the "values" of the 3×3 matrix surrounding the pixel. In the Hue-Saturation-Value color model, the "value" is the maximum of the normalized red, green, and blue values of the pixel; conceptually, the brightness of that pixel. The new value for the pixel is: $\frac{\text{value} - p*(\text{mean value})}{1-p}$, where p is a value between 0 and 1. The filter exaggerates the contrast between a pixel's brightness and the average brightness of the surrounding pixels.

2. Despeckle Filter: which replaces each pixel with the median color in a 3×3 matrix surrounding the pixel. Used for noise reduction, the algorithm gathers the colors in the square matrix, sorts them using an inlined Shell sort, and chooses the median element.

3. Edge Detect Filter: which runs a merging of a pair of convolutions, one that detects horizontal edges, and one that detects vertical edges. The convolution is done separately for each plane (red, green, blue) of the image, so where there are edges in the red plane, for example, the resultant image will highlight the red edges.

4. Emboss filter: which applies a 3×3 convolution matrix to the image, a variation of an edge detection algorithm. Most of the image is left as a medium gray, but leading and trailing edges are turned lighter and darker gray, respectively.

5. Oil Paint Filter: which computes a histogram of a 3×3 matrix surrounding the pixel and chooses the most frequently occuring color in the histogram to supplant the old pixel value. The result is a localized smearing effect.

6. Pixelize Filter: which replaces each pixel in each 4×4 squares in the image with the average color in the square matrix.

7. Spread Filter: which replaces each pixel by a random one within a 3×3 matrix surrounding the pixel.

Figure 8 illustrates the original image and processed images that result from applying four of the filters described above. Although some of these filters are not necessarily useful in the medical domain, they follow the same pattern of spatial image processing: the traversal or convolution of a fixed size or variable size matrix over pixels surrounding each pixel in the image array. In principle, therefore, the performance of this set of filters reflect the performance of other more relevant filters of comparable complexity.

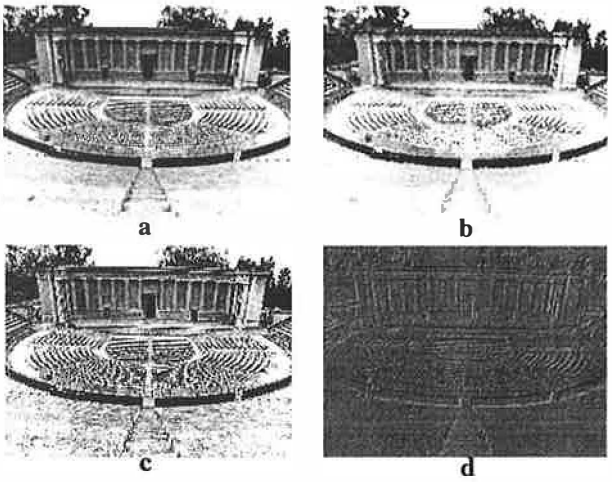


Figure 8: (a) Original Image; (b) Oil-painted Image; (c) Sharpened Image; (d) Embossed Image

3.1.2 Performance Metrics

We measured the performance of MedJava in comparison with the NT port of the xv subset by sending an 8-bit image at equidistant degrees of magnifications through each of eight filters 10 times, keeping the average of the trials. Both xv and Java convert 8-bit images, either greyscale or color, into 24-bit RGB color images prior to filtering. Moreover, all eight algorithms are functions solely of image dimension and not pixel value. Thus, there is no processing performance difference between greyscale and color images in either environment.

We expected *a priori* that the C code would out perform the Java filters due to the extensive optimizations performed by the Microsoft Visual C/C++ compiler.[3] Therefore, we coded the MedJava image filters using the source level optimization techniques described in Section 3.1.3 to elicit maximum performance from them.

However, contrary to our expectations, the hand optimized Java algorithms performed nearly as well as their C counterparts. Therefore, we also optimized the C algorithms by hand. This rendered the two sets of algorithms nearly indistinguishable in appearance, but not indistinguishable in performance. For MedJava, we ran three trials, one on Internet Explorer 4.0 release 2 on NT (IE 4), one on Netscape Navigator 4.0 on NT (NS 4), and one on Internet Explorer 4.0 release 2 with just-in-time compilation disabled (IE 4 JIT off).

3.1.3 Source Level Optimizations

The Java run-time system, including the garbage collector and the Abstract Window Toolkit (AWT), was written using C. Therefore, they cannot be optimized by the Java bytecode interpreter or compiler. As a result, any attempt to improve the performance of Java in medical imaging systems must improve the performance of code spent outside these areas, *i.e.*, in the image filters themselves.

JIT compilers affect the greatest speed up in computationally intensive tasks that do not call the AWT or run-time system, as shown by the benchmarks in Table 1. These benchmarks test the performance of common image filter operations in the two browsers used in the experiments. These data were obtained by wrapping a test harness around a loop that iterates for a fixed, but large, number of iterations, subtracting the loop overhead from the result, and dividing by the number of iterations. Java's garbage collection routine was called before

[3] "Full optimization" on MVC++ includes: inline function expansion, subexpression elimination, automatic register allocation, loop optimization, inlining of common library functions, and machine code optimization.

operation	NS 4	IE 4	IE 4 JIT off
Loop overhead	10.21	10.21	902
Quick Int Assignment	5.01	5.01	339
Local Int Assignment	5.01	5.32	440
Static Member Integer	25.24	20.13	851
Member Integer	5.01	10.02	560
Reference Assignment	5.01	10.12	580
Integer Array Access	11.63	5.21	350
Static Instance Method	35.45	34.95	692
Instance Method	40.46	30.24	922
Final Instance Method	30.35	40.17	922
Private Instance Method	35.46	30.24	992
Random.nextInt()	80.92	86.71	450
int++	20.33	7.72	180
int = int + int	5.02	10.11	710
int = int - int	15.12	10.12	690
int = int * int	10.12	10.12	691
int = int / int	75.96	71.35	890
int /= 2	16.43	10.92	931
int >>= 1	17.43	7.21	661
int *= 2	19.63	7.42	780
int <<=1	17.33	7.71	731
int = int & int	15.13	5.01	670
int = int | int	5.01	0.11	670
float = float + float	15.12	10.12	730
float = float - float	15.12	10.12	700
float = float * float	10.02	15.22	720
float = float / float	46.82	46.02	841
Cast double to float	4.91	5.01	570
Cast float to int	67.14	347.3	910
Cast double to int	67.15	13.06	920

Table 1: Times in Nanoseconds for Common Operations in the Testbed Java Environment

the sequence to prohibit it from affecting the test results. The results are listed in nanoseconds.

Since the conversion from byte-code to native code is already costly, JIT compilers do not spend a great deal time attempting to further optimize the native code. Therefore, lacking source to a bytecode compiler that optimizes its output, the most a developer of performance-critical applications can do to further accelerate the performance of computationally-intensive tasks is to optimize the source code manually. The image filters in MedJava leveraged the following canonical techniques and insight on how to best optimize computationally intensive source code in Java [19]:

Strength reduction: which replaces costly operations with a faster equivalent. For instance, the Image Filters converted multiplications and divides by factors of two into lefts and rights shifts.

Common subexpression elimination: which removes redundant calculations. Image Filters store the pixel values in a one dimensional array. Thus, for each pixel access this optimization calculates a pixel index once from the column and row values and stores the results of the array access into a temporary variable, rather than continually indexing the same element of the array.

Code motion: which hoists constant calculations out of loops. Thus, although it may be impossible to unroll loops where the number of iterations is a function of the image height and width, the Image Filters reduce the overhead of loops by removing constant calculations computed at each loop termination check.

Local variables: which are efficient to access. The virtual machine stores them in an array in the method frame. Thus, there is no overhead associated with dereferencing an object reference, unlike an instance variable, a class name, or a static data member. The bytecodes `getfield` and `getstatic` must first resolve the class and method names before pushing the value of the variable onto the operand stack. Also, the `iload` and `istore` instructions allow the JVM to quickly load and store the first four local variables to and from the operand stack [20].

Integer variables, floats, and object references: which are most directly supported by the JVM since the operand stack and local variables are each one word in width, the size of integers, floating points, and references. Smaller types, such as `short` and `byte` are not directly supported in the instruction set. Therefore, each must be converted to an `int` prior to an operation and then subsequently back to the smaller type, accruing the cost of a valid truncation [20].

Manually inlining methods: eliminates the overhead associated with method invocation. Although `static`, `final`, and `private` methods can be resolved at compile time, eliminating method calls entirely, especially simple calls on the `java.lang.Math` package (*e.g.*, `ceil`, `floor`, `min`, and `max`), in critical sections of looping code will further improve performance.

The `final`, `static`, or `private` keywords on a method advises the run-time compiler or interpreter that it may safely inline the method. However, because classes are linked together at run-time, changes made to a `final` method in one class would not be reflected in other already compiled classes that invoke that method, unless they too were recompiled [21]. Naturally, when invoking methods internal to a class, this is not a problem. Moreover, the `-O` option on the Sun `javac` source to bytecode compiler requests that it attempt to inline methods.

As an example of worthwhile manual method inlining, an `ImageFilter` contains a method called `setColorModel`. In this method the `ImageProducer` provides the `ImageFilter` with the `ColorModel` subclass that grabs the color values of each pixel in the image source. `ColorModel` is an abstract class with methods `getRed`, `getGreen`, and `getBlue` to retrieve the specific color value from each pixel. Thus, every time a filter needs a color value, it must incur the overhead of this dynamically resolved method call. However, calling the `getDefaultColorModel` static method on `ColorModel` returns a `ColorModel` subclass, which guarantees that each pixel will be in a known form, where the first 8 bits are the alpha (transparency) value, the next 8 are the red, the next 8 are the green, and the last 8 bits are the blue value. Therefore, rather than using the methods on `ColorModel` to retrieve the color values, the `ImageFilter` can retrieve values simply by shifting and masking the integer value of the pixel, *e.g.*, to obtain the red value of a pixel: (pixel >> 16) & 0xff.

Of course, for C code many of the same optimization techniques apply. We ran a similar set of operation benchmarks in C, using the same test harness technique as we did for the Java benchmarks. The results, shown in Table 2, are the mean of 5 trials, with each measurement exhibiting a standard deviation of no more than 0.5 nanoseconds.

With "global optimizations" enabled, the MSVC++ compiler will actively assign variables to registers at its own discretion. With optimizations disabled, it takes no special measures to abide by the `register` keyword. Again, the results are listed in nanoseconds.

Table 2 reveals that the MSVC++ generated code yields comparable performance with the output of the two JIT compilers. Narrowing casts, for example from floating point to integer data is more time consuming in the MSVC++ generated code than the JIT output, however, calls to static, external, and library functions (*e.g.*, `rand`) are less time consuming than their Java method equivalents. Also, floating point multiplication and division, translated into the `fmul` and `fdiv` in MSVC++, lag behind the JIT translation of these operations.

3.1.4 Performance Results and Evaluation

Figures 9–16 plot our results for each of the eight filters on each of the three language/compiler permutations.

Using insights about the most frequently performed operations in the algorithms, and the tables enumerating the costs of those operations on the three configurations (Tables 1 and 2), we can attempt to explain any observed, counter-intuitive differences in the performance of the algorithms.

In general, the hand-optimized Java algorithms executed in times comparable with their C hand-optimized counterparts. However, the added benefit of the MSVC++ compile-time op-

operation	MSVC++ output
Local int access	7.40
Extern int access	3.57
Static int access	7.13
Heap byte access	7.62
Heap int access	7.99
Stack byte array	7.72
Stack int array	7.82
Global byte array	8.60
Global int array	8.17
Static function call	25.40
Extern function call	32.02
int++	7.39
int = int + int	11.88
int = int - int	11.88
int = int * int	21.88
int = int / int	203.27
int *= 2	7.18
int <<=1	3.55
int /= 2	13.47
int >>= 1	7.25
int = int & int	11.86
int = int | int	7.68
float = float + float	16.58
float = float - float	16.51
float = float * float	556.08
float = float / float	611.43
Call to rand()	57.33
cast from float to int	863.11

Table 2: Times in Nanoseconds for Common Operations in C/C++ on the Testbed Platform

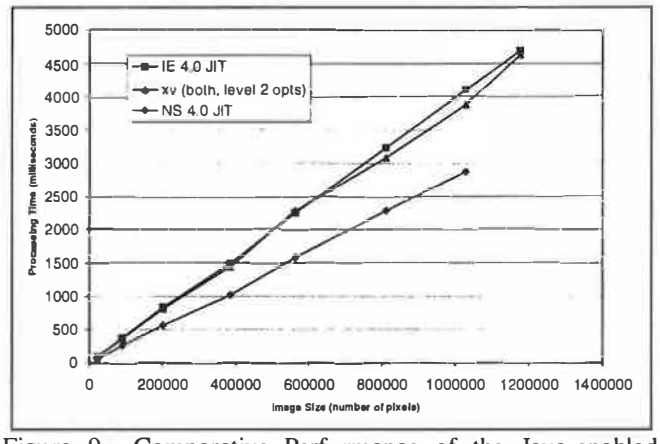


Figure 9: Comparative Performance of the Java-enabled Browsers and `xv` in Applying the Sharpen Image Filter to an Image at Various Sizes

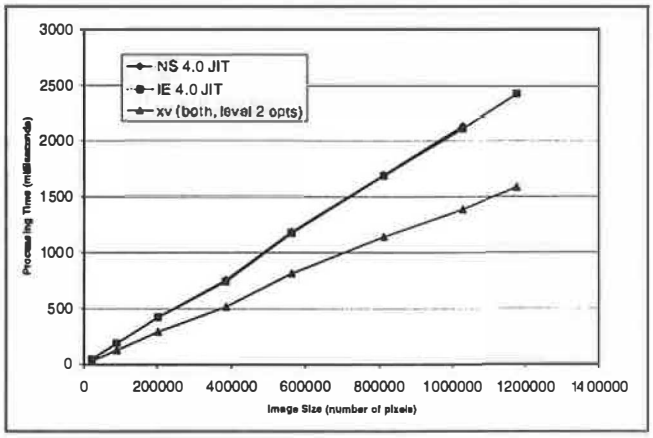


Figure 10: Comparative Performance of the Java-enabled Browsers and xv in Applying the Edge Detection Image Filter to an Image at Various Sizes

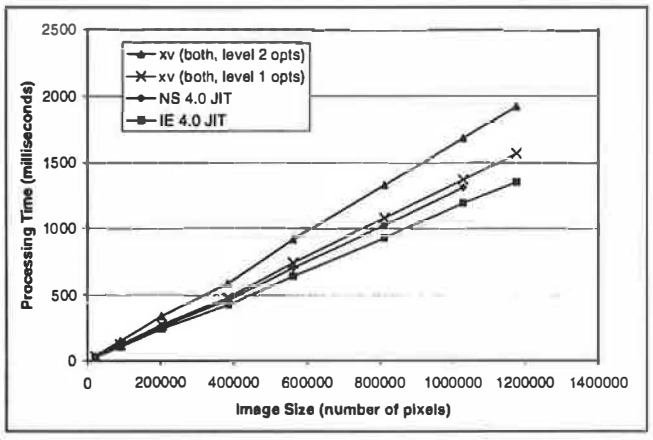


Figure 11: Comparative Performance of the Java-enabled Browsers and xv in Applying the Blur Image Filter to an Image at Various Sizes

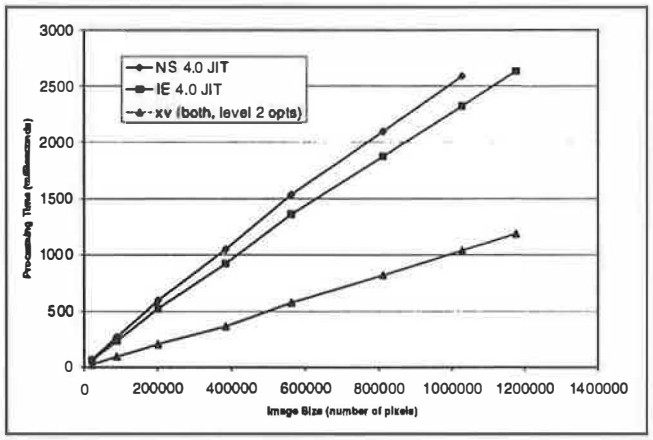


Figure 12: Comparative Performance of the Java-enabled Browsers and xv in Applying the Despeckle Image Filter to an Image at Various Sizes

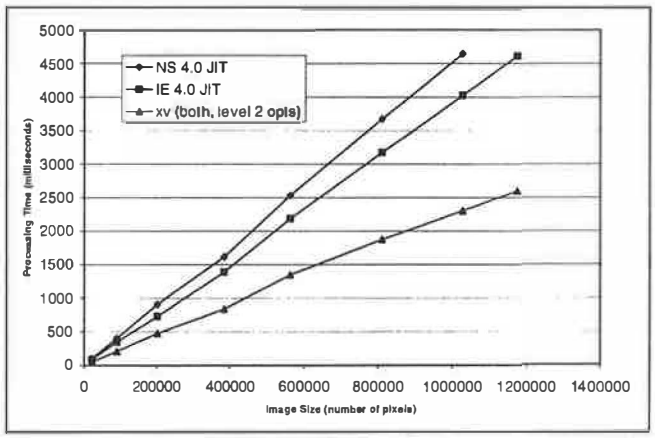


Figure 13: Comparative Performance of the Java-enabled Browsers and xv in Applying the Oil Paint Image Filter to an Image at Various Sizes

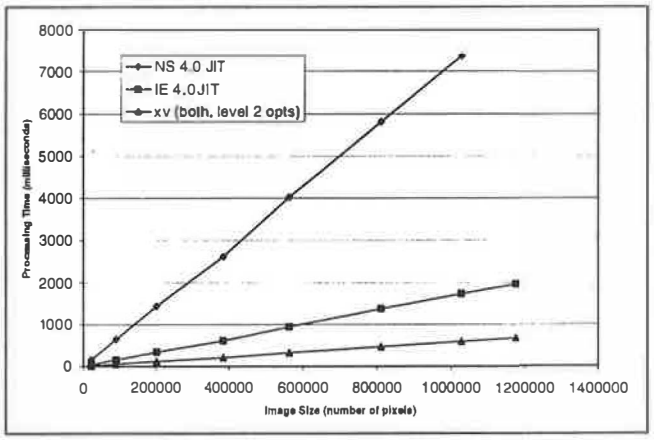


Figure 14: Comparative Performance of the Java-enabled Browsers and xv in Applying the Spread Image Filter to an Image at Various Sizes

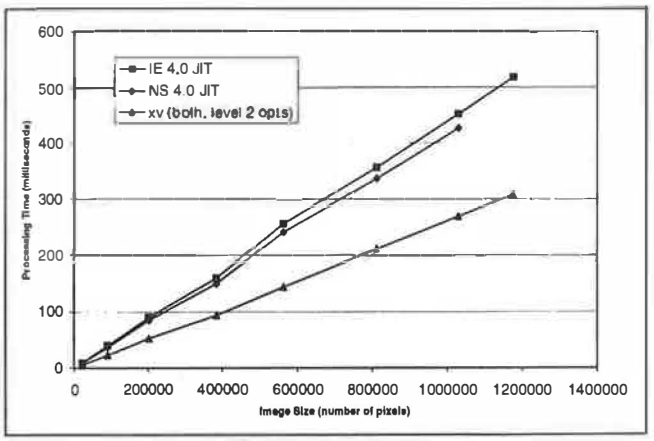


Figure 15: Comparative Performance of the Java-enabled Browsers and xv in Applying the Emboss Image Filter to an Image at Various Sizes

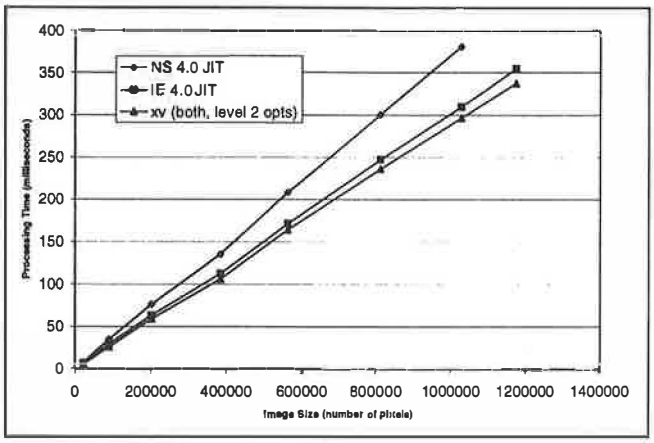


Figure 16: Comparative Performance of the Java-enabled Browsers and xv in Applying the Pixelize Image Filter to an Image at Various Sizes

timizations gave the C algorithms a competitive advantage. Thus, for all but two of the filters (Blur image filer and Sharpen image filter), the C algorithms outperformed their Java equivalents.

Contributing to the overall superiority of the C algorithm execution is fast array access time and increments, induced by the rigorous utilization of registers by the compiler. However, applying the techniques of code movement and strength reduction help the Java code to negate any benefits of similar compile-time optimization performed by the C/C++ compiler.

There is one severely aberrational case in which the C runtime performed more poorly than the Java ones: the sharpen filter. In the sharpen filter, color values are continually converted between the integer RGB format and the floating point HSV format. Netscape, whose floating point to integer narrowing conversion performance exceeds Internet Explorer's and C/C++'s, has the competitive advantage.

3.2 High-speed Network Benchmarking

As described earlier, high performance is one of the key forces that guides the development of a distributed EMIS. In particular, it is important that medical images be delivered to radiologists and processed in a timely manner to allow proper diagnosis of patients exams. To evaluate the performance of Java as a transport interface for exchanging large images over high-speed networks, we performed a series of network benchmarking tests over ATM. This section compares the results with the performance of C/C++ as a transport interface [22].

3.2.1 Benchmarking Configuration

Benchmarking testbed: The network benchmarking tests were conducted using a FORE systems ASX-1000 ATM switch connected to two dual-processor UltraSPARC-2s running SunOS 5.5.1. The ASX-1000 is a 96 Port, OC12 622 Mbs/port switch. Each UltraSparc-2 contains two 168 MHz Super SPARC CPUs with a 1 Megabyte cache per-CPU. The SunOS 5.5.1 TCP/IP protocol stack is implemented using the STREAMS communication framework. Each UltraSparc-2 has 256 Mbytes of RAM and an ENI-155s-MF ATM adaptor card, which supports 155 Megabits per-sec (Mbps) SONET multimode fiber. The Maximum Transmission Unit (MTU) on the ENI ATM adaptor is 9,180 bytes. Each ENI card has 512 Kbytes of on-board memory. A maximum of 32 Kbytes is allotted per ATM virtual circuit connection for receiving and transmitting frames (for a total of 64 K). This allows up to eight switched virtual connections per card.

Performance metrics: To evaluate the performance of Java, we developed a test suite using Java ACE. To measure the performance of C/C++ as a transport interface, we used an extended version of TTCP protocol benchmarking tool [22]. This TTCP tool measures the throughput of transferring untyped bytestream data (*i.e.*, Blobs [14]) between two hosts. We chose untyped bytestream data, since untyped bytestream traffic is representative of image pixel data, which need not be marshaled or demarshaled.

3.2.2 Benchmarking Methodology

We measured throughput as a function of sender buffer size. Sender buffer size was incremented in powers of two ranging from 1 Kbytes to 128 Kbytes. The experiment was carried out ten times for each buffer size to account for variations in ATM network traffic. The throughput was then averaged over all the runs to obtain the final results.

Since Java does not allow manipulation of the socket queue size, we had to use the default socket queue size of 8 Kbytes on SunOS 5.5. We used this socket queue size for both the Java and the C/C++ network benchmarking tests.

3.2.3 Performance Results and Evaluation

Throughput measurements: Figure 17 shows the throughput measurements using Java and C/C++ as the transport interface. These results illustrate that the C/C++ transport interfaces consistently out-perform the Java transport interfaces. The performance of both the Java version and the C/C++ version peak at the sender buffer size of 8 Kbytes. This result stems from the fact that 8 Kbytes is close to the MTU size of the ATM network, which is 9,180 bytes. The results indicate that for a sender buffer size of 1 Kbytes. C/C++ outperforms Java by only about 2%. On the other hand, for sender buffer size of 2 Kbytes, C/C++ out-performs Java by more than 50%. C/C++ out-performs Java by 15%-20% for the remaining sender buffer sizes.

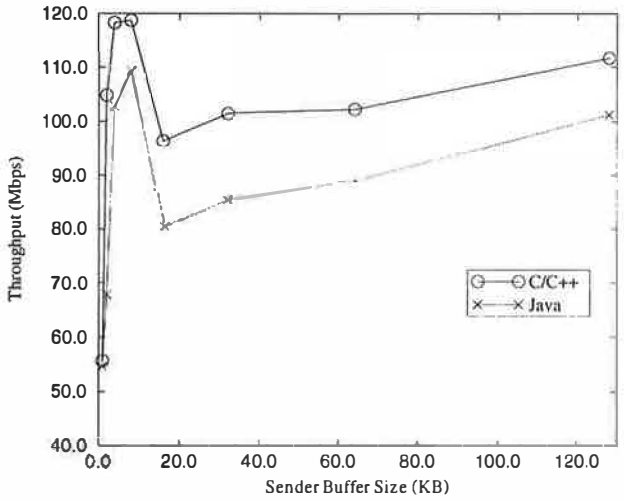


Figure 17: Throughput Measurement of Using Java and C/C++ as the Transport Interface

Analysis summary: C/C++ out-performed Java as the transport interface for all sender buffer sizes. The difference in the performance between Java and C/C++ is reflective of the overhead incurred by the JVM. This overhead can be either in the form of interpreting Java byte code (if an interpreter is used) or in the form of compiling Java byte code at run time (if a JIT compiler is used).

However, it is important to note that despite the differences in performance between Java and C/C++, Java performs comparably well. A throughput of about 110 Mbps on a 155 Mbps ATM network is quite efficient considering the default socket queue size is only 8 Kbytes. Results [23] show that network performance can improve significantly if the maximum socket queue size (64 Kbytes) is used. If Java allowed programmers to change the socket queue size the throughput should be higher for larger sender buffer sizes.

4 Related Work

Several studies have measured the performance of Java relative to other languages. In addition, many techniques have been proposed to improve the performance of Java. The following is a summary of the related work in this area.

4.1 Measuring Java's Performance

Several studies have compared the execution time of Java interpreted code and Java compiled code with the execution time of C/C++ compiled code. Shiffman [24] has measured and compared the performance of several programs written both in Java and in C++. For the tests performed, Java interpreted code performed 6 to 20 times slower than compiled C++ code, while Java compiled code performed only about 1.1 to 1.5 times slower than C++ code.

The results obtained in [24] differ from the ones we obtained because the tests run were also different. The tests carried out by Shiffman involved measuring the timings for iterative and recursive versions of a calculator of numbers in the Fibonacci series, as well as a calculator of prime numbers. The results, however, once again indicate that the Java code performs reasonably well, compared with C/C++ code. This finding is consistent with our results for the sv image processing algorithms.

4.2 Improving Java's Performance

Several groups are working on improving the performance of JIT compilers, as well as developing alternatives to JIT compilers.

Toba: A system for generating efficient stand-alone Java applications has been developed at the University of Arizona [25]. The system is called Toba and generates executables that are 1.5 to 4.4 times faster than alternative JVM implementations. Toba is a "Way-Ahead-of-Time" compiler and therefore converts Java code into machine code before the application is run. It translates Java class files into C code and then compiles the C code into machine code making several optimizations in the process. Although such a compiler can be very useful for stand-alone Java applications, it can not, unfortunately, be used for Java applets.

Harrisa: An efficient environment for the execution of Java programs called Harissa has been developed at the University of Rennes [26]. Harissa mixes compiled and interpreted code. It translates Java bytecode to C and in the process makes several optimizations. The resulting C code produced by Harissa is up to 140 times faster than the JDK interpreter and 30% faster than the Toba compiler described above.

Unlike Toba, Harissa can work with Java applets also. Therefore, Harissa can be used by MedJava to improve the performance of image processing and bringing it closer to the performance of a similar application written in C/C++.

Asymetrix: Another approach similar to Harissa is SuperCede VM developed by Asymetrix [27]. SuperCede is a high-performance JVM that can improve the performance of Java to execute at native C/C++ speed. Unlike JIT compilers, where the interpreter selectively compiles functions, SuperCede compiles all class files as they are downloaded from the server. The result is an application that is fully compiled to machine code and can therefore execute at native C/C++

speed. SuperCede VM can also work with Java applets and can therefore be used by MedJava to improve its performance.

4.3 Evaluating Web Browsers Performance

Several studies compare the performance of different Web browsers.

CaffeineMark: Pendragon Software [28] provides a tool called that can be used for comparing different Java virtual machines on a single system, *i.e.,* comparing appletviewers, interpreters and JIT compilers from different vendors. The CaffeineMark benchmarks measures Java applet/application performance across different platforms. CaffeineMark benchmarks found Internet Explorer 3.01 on NT to contain the fastest JVM followed by Internet Explorer 3.0 on NT. Netscape Navigator 3.01 on NT performed sixth in their tests. Unfortunately, the CaffeineMark benchmarks do not include the latest versions of the Web browsers that we used to run our tests, *i.e.*, Internet Explorer 4.0 and Netscape 4.0. Therefore, their results are out-of-date.

PC Magazine: Java performance tests in PC Magazine reveal the strengths and flaws of several of today's Java environments [29]. Their tests reveal significant performance differences between Web browsers. In all their tests, browsers with JIT compilers out-perform browsers without JIT compilers by up to 20 times. This is consistent with the results we obtained. Their tests found Internet Explorer 3.0 to be the fastest Java environment currently available. They found Netscape Navigator 3.0 to be consistently slower than Internet Explorer. Once again their tests did not make use of the latest versions of the Web browsers and therefore are out-of-date.

5 Concluding Remarks

This paper describes the design and performance of a distributed electronic medical imaging system (EMIS) called MedJava that we developed using Java applets and Web technology. MedJava allows users to download images across the network and process the images. Once an image has been processed, it can be uploaded to the server where the applet was downloaded.

The paper presents the results of systematic performance measurements of our MedJava applet. MedJava was run in two widely-used Web browsers (Netscape and Internet Explorer) and the results were compared with the performance of `xv`, which is an image processing application written in C. In addition, the paper presented performance benchmarks of using Java as a transport interface to transfer large images over high-speed ATM networks.

The following is a summary of the lessons learned while developing MedJava:

Compiled Java code performs relatively well for image processing compared to compiled C code: In our image processing tests, interpreted Java code was substantially out-performed by compiled Java code and compiled C code. The image processing application written in C out-performed MedJava in most of our tests. However, only when the C code was itself hand-optimized, were the MSVC++ compiler's compile-time optimizations able to produce significantly more efficient code. If techniques become available to employ such optimization techniques in JIT compilers without incurring unacceptable latency, then this advantage will be abated. In addition, efficient Java environments like Harissa, which mix byte code and compiled code, can further improve the performance of Java code and allow it to perform as well as the performance of C code.

Compiled Java code performs relatively well as a network transport interface compared to compiled C/C++ code: Our network benchmarks illustrate that using C/C++ as the transport interface out-performs using Java as the transport interface by 2% to 50%. The difference of 50% in performance between Java and C/C++ for a buffer size of 2 KB occurs because of a sudden jump in the throughput in the case of C/C++ in going from a sender buffer size of 1 KB to a sender buffer size of 2 KB. In the case of C/C++, throughput jumped from 55.69 Mbps to 104.81 Mbps in going from a sender buffer size of 1 KB to a sender buffer size of 2 KB. In the case of Java, however, the increase in throughput was gradual and therefore resulted in a large performance difference for sender buffer size of 2 KB.

The performance of using Java as the transport interface peaks at the sender buffer size close to the network MTU size and is only 9% slower than the performance of using C/C++ as the transport interface. Therefore, Java is relatively well-suited to be used as the transport interface.

It is becoming feasible to develop performance-sensitive distributed EMIS applications in Java: The built-in support for GUI development, the support for image processing, the support for sockets and threads, automatic memory management, and exception handling in Java simplified our task of developing MedJava. In addition, the availability of JIT compilers allowed MedJava to perform relatively well compared to a applications written in C/C++.

Therefore, we believe that it is becoming feasible to use Java to develop performance-sensitive distributed EMIS applications. In particular, even when Java code does not run quite as fast as compiled C/C++ code, it can still be a valuable tool for building distributed EMISs because it facilitates rapid prototyping and development of portable and robust applications.

Netscape 4.0 is the fastest Java environment currently available: Among the Web browsers, those providing JIT compilers in the JVM clearly out-perform browsers that do not provide JIT compilers. Both Internet Explorer 4.0 and Netscape 4.0 running Windows NT on the Intel instruction set provide JIT compilers in their JVMs. However, in several cases, Netscape 4.0 on NT performed more than twice as fast as Internet Explorer 4.0 on NT . Therefore, among Java-enabled Web browsers, Netscape 4.0 is the fastest Java environment currently available for image processing.

Java has several limitations that must be fixed to develop production distributed EMIS: Even though Java resolves several of the forces of developing a distributed EMIS, it still has the following limitations:

- **Memory limitations:** We found that applying image filters to images larger than 1 MB causes the JVM of both Netscape 4.0 on NT and Internet Explorer 4.0 on NT to run out of memory. This can hinder the development of distributed EMISs since many medical images are larger than 1 MB.

- **Lack of AWT portability:** We found the AWT implementations across platforms to be inconsistent, thereby making it hard to develop a uniform GUI. When we tried running MedJava on different brower platforms, we found some features of MedJava do not work portably on certain platforms due to lack of support in the JVM where the applet was run.

- **Security impediments:** We found the lack of ability to upload images to servers other than the one from where the applet was downloaded from as another significant limitation of using Java for distributed EMISs. Although these restrictures were added to Java as a security-feature of applets, they can be quite limiting. The following are several workarounds for these security restrictions:

  - One approach is to run a CGI Gateway at the server from where the MedJava applet is downloaded. MedJava can then make uploading requests to the Gateway that can then upload images to servers across the network.
  - Another scheme can be used to solve this problem without requiring an additional Gateway to run at the server. This requires adding a security authentication mechanism within the Java `Applet` class. This mechanism can then allow an applet to upload files to servers other than the one from where the applet was downloaded. Java version 1.1 allows an applet context to download signed Java archive files (JARs), which contain Java classes, images, and sounds. If these are signed by a trusted entity using its private key, applets can run in the context with the full privileges of local applications. The context uses the public key for a entity, authenticated by a Certificate from another trusted entity, to verify that the archive file came from a trusted signer. Therefore, an EMIS signed by a trusted entity could run in a browser with the ability to save files to the local file system and open network connections to machines other than the one from which it was downloaded.

In summary, our experience suggests that Java can be very effective in developing a distributed EMIS. It is simple, portable and distributed. In addition, compiled Java code can be quite efficient. If the current limitations of Java are resolved and highly-optimizing Java compilers become available, it should be feasible to develop performance-sensitive distributed applications in Java.

The complete source code for Java ACE is available at `www.cs.wustl.edu/~schmidt/JACE.html`.

Acknowledgments

We would like to thank Karlheinz Dorn and the other members of the Siemens Medical Engineering group in Erlangen, Germany for their technical support, sponsorship, and friendship during the MedJava project.

References

[1] W. L. R. J. J. Conklin, "Digital Management and Regulatory Submission of Medical Images from Clinical Trials: Role and Benefits of the Core Laboratory," *Proc. SPIE, Health Care Technology Policy II: The Role of Technology in the Cost of Health Care: Providing the Solutions*, vol. 2499, Oct. 1995.

[2] G. Blaine, M. Boyd, and S. Crider, "Project Spectrum: Scalable Bandwidth for the BJC Health System," *HIMSS, Health Care Communications*, pp. 71–81, 1994.

[3] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging," *USENIX Computing Systems*, vol. 9, November/December 1996.

[4] F. L. Kitson, "Multimedia, Visual Computing, and the Information Superhighway," *Proc. SPIE, Medical Imaging 1996: Image Display*, vol. 2707, Apr. 1996.

[5] P. Jain and D. Schmidt, "Experiences Converting a C++ Communication Software Framework to Java," *C++ Report*, vol. 9, January 1997.

[6] P. Jain and D. C. Schmidt, "Service Configurator: A Pattern for Dynamic Configuration of Services," in *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, USENIX, June 1997.

[7] A. Wollrath, R. Riggs, and J. Waldo, "A Distributed Object Model for the Java System," *USENIX Computing Systems*, vol. 9, November/December 1996.

[8] *Java API Documentation Version 1.0.2*. Available from http://java.sun.com:80/products/jdk/1.0.2/api.

[9] M. P. Plezbert and R. Cytron, "Does Just in Time = Better Late than Never?," in *ACM 1997 Symposium on the Principles of Programming Languages*, 1997.

[10] H. R. Myler and A. R. Weeks, *Computer Imaging Recipes in C*. Prentice Hall, Inc. Englewoods Cliffs, New Jersey, 1993.

[11] J. Hu, S. Mungee, and D. C. Schmidt, "Principles for Developing and Measuring High-performance Web Servers over ATM," in *Proceeedings of INFOCOM '98*, March/April 1998.

[12] J. Hu, I. Pyarali, and D. C. Schmidt, "Measuring the Impact of Event Dispatching and Concurrency Models on Web Server Performance Over High-speed Networks," in *Proceedings of the 2nd Global Internet Conference*, IEEE, November 1997.

[13] *Java ACE Home Page*. Available from http://www.cs.wustl.edu/ schmidt/JACE.html.

[14] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging," in *Proceedings of the 2nd Conference on Object-Oriented Technologies and Systems*, (Toronto, Canada), USENIX, June 1996.

[15] D. C. Schmidt and T. Suda, "An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems," *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, vol. 2, pp. 280–293, December 1994.

[16] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Initializing Communication Services," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[17] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.

[18] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley and Sons, 1996.

[19] D. Bell, "Make Java fast: Optimize!." JavaWorld, April 1997. Available from http://www.javaworld.com/javaworld/jw-04-1997/jw-04-optimize.html.

[20] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[21] J. Gosling, B. Joy, and G. Steele, *The Java Programming Language Specification*. Reading, MA: Addison-Wesley, 1996.

[22] A. Gokhale and D. C. Schmidt, "Measuring the Performance of Communication Middleware on High-Speed Networks," in *Proceedings of SIGCOMM '96*, (Stanford, CA), pp. 306–317, ACM, August 1996.

[23] D. C. Schmidt, T. H. Harrison, and E. Al-Shaer, "Object-Oriented Components for High-speed Network Programming," in *Proceedings of the 1st Conference on Object-Oriented Technologies and Systems*, (Monterey, CA), USENIX, June 1995.

[24] H. Shiffman, "Boosting Java Performance: Native Code & JIT Compilers." Available from http://reality.sgi.com/shiffman/Java-JIT.html, 1996.

[25] T. A. Proebsting, G. Townsend, P. Bridges, J. H. H. T. Newsham, and S. Watterson, "Toba: Java For Applications, A Way Ahead of Time (WAT) Compiler," in *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, 1997.

[26] G. Muller, B. Moura, F. Bellard, and C. Consel, "Harissa: A Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code," in *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, 1997.

[27] Asymmetrix, "SuperCede." Available from http://www.asymetrix.com/products/supercede/index.html, 1997.

[28] P. Software, "CaffeineMark(tm) 2.5: The Industry Standard Java Benchmark." Available from http://www.webfayre.com/pendragon/cm2/index.html, 1996.

[29] R. V. Dragan and L. Seltzer, "Java Speed Trials." Available from http://www8.zdnet.com/pcmag/features/pctech/1518/java.htm, 1996.

Dynamic Management of CORBA Trader Federation

Djamel Belaïd, Nicolas Provenzano, Chantal Taconet[1]
Institut National des Télécommunications
Evry, France

Abstract

In Wide Area Networks, tools for discovering objects that provide a given service, and for choosing one out of many are essential. The CORBA trading service is one of these tools. A trader federation extends the limit of a discovery, thanks to the cooperation of several trading servers. However, in a federation, cooperation links are manually and statically established. In this article, we propose an Extended Trading Service, which manages a trader federation dynamically. With Extended Trading Service, optimized links between traders are automatically set up thanks to the use of a minimum-weight spanning tree. Cycles in discovery propagation are eliminated. Links evolve dynamically in order to adapt to the modification of the underlying topology and the failure of an intermediate trader or link. The Extended Trading Service is able to organize discovered objects from the nearest to the furthest according to a distance function chosen for the federation. Such management is provided by specialized trading servers conforming to OMG Trading Service Specification.

1. Introduction

The expansion of Wide Area Networks (**WANs**) has already led to information superhighways. From now on, a huge number of computer services are being developed and made available on WANs. These services should be available from a wide range of computer stations and through a wide range of networks. Object middleware, such as those built with *CORBA* (Common Object Request Broker Architecture) [OMG97], help to implement distributed services without taking care of distribution configuration.

In WANs, huge number of clients, and distances between clients, have naturally led to replicate some server objects on geographically distributed networks. Every day, new replicas may appear. One issue is to offer final users tools for transparent discovering services and in the case of replicated servers, for discovering the "best" one, which may be different for each client.

Tools currently offered to find out server objects are not adapted for finding the "best" server for each client. Traditional name servers such as *DNS* [Mock87], compel one to give different names to different replicas (e.g. different *URLs* [Bern94]) and so don't help end users to choose the best server. The *DNS* support for replication [Bris95] automatically selects a server through a round robin algorithm and as a result the chosen server is not adapted to each client. Discovery tools, such as *Alta Vista Search Engine* [Seit96], offer global searching on the Internet but selection is on text information only.

Some new kinds of discovery tools are appearing. With the *Globe* location service, servers are registered in a global hierarchical tree [vS96], which is then used for finding the nearest server. The Internet community is thinking of *URN (Uniform Resource Name)* [Moat97] to replace *URL* in order to have better support for replicated and movable servers. The trading service specification, proposed for *ODP* [ODP93] and then *CORBA* [OMG96], has been defined to find the best server(s) for each client thanks to a service type and a list of properties.

In WANs, some services are available thanks to the cooperation of a set of distributed servers. We illustrate this feature through the following examples. The *News USENET* [Kant86] is distributed to final users thanks to the cooperation of several news servers distributed on the Internet. In the *MBone* [Deer90], the

[1] INT, 9 rue Charles Fourier, 91 011 Evry Cedex, France.
{Djamel.Belaid, Nicolas.Provenzano, Chantal.Taconet}@int-evry.fr

subset of Internet supporting multicast routing, packets are forwarded thanks to tunnels set up between a set of cooperating multicast routers. And a federation of cooperating traders may offer the trading service. The study of these examples shows that cooperation links between these servers are set up manually by human administrators on each server. These cooperation links are neither necessarily adapted to the underlying network topology nor fault tolerant. One issue is to offer a tool for linking cooperating servers efficiently and dynamically.

The *CORBA* trading specification does not offer any tool for linking cooperating traders dynamically either. In this article, we define a specialized trader, conforming to the *OMG* specification, which offers dynamic management of trader federation. Our federation is based on the cooperating server graph model [Taco97] that optimizes the links set up between cooperating traders and helps them to find the nearest server to each client.

This article is organized as follows. In Section 2, we present a synthesis of the *CORBA* trading service specification and we study limitations of this service in the WAN context. In Section 3, we summarize the Cooperating Server Graph model that offers to manage links between cooperating servers dynamically. We use this model in Section 4 to define the architecture of a specialized trader for WANs. In Section 5 we study its implementation and compare its behavior with traditional traders in the WAN context.

2. CORBA Trading Service description

In this section, after a brief description of the *CORBA* architecture, we present the main features of the CORBA Trading Service, and then some optimizations of this service intended to the WAN context.

2.1. CORBA Architecture

Common Object Request Broker Architecture (*CORBA*) [OMG97] is an open distributed object computing infrastructure standardized by the *Object Management Group* (*OMG*). *CORBA* allows development of applications in which distributed objects communicate with one another thanks to well-defined interfaces, no matter where objects are located or how objects are implemented.

In *CORBA* each object is identified by an **object reference** and is associated to an interface and an implementation. An interface allows clients to access a set of services offered by a server object. Interfaces are described with the *CORBA Interface Description Language* (*CORBA-IDL*).

CORBA Architecture consists of the following components:

- **Object Request Broker** (*ORB*) is a middleware that establishes client-server interaction between distributed objects. When a client invokes a method on a server, whatever programming language or operating system used for server implementation, *ORB* has to find the server implementation location, deliver the request to this server, and return invocation results to the client. In order to allow interaction between objects on different *ORBs*, *OMG* has defined a *General Inter-ORB Protocol* (*GIOP*), which specifies a standard transfer syntax and protocol. *GIOP* is designed to operate over a connection-oriented transport protocol. *Internet Inter-ORB Protocol* (*IIOP*) is a concrete implementation of the abstract *GIOP* for *TCP/IP*.

- **Object Services**, is a collection of services that provide basic, nearly system-level, functions for implementing objects. We can mention Naming Service, Life Cycle Service, and Trading Service.

- **Common Facilities**, is a collection of services shared by many applications. For example, graphical objects may be used by every application for providing a user interface.

- **Applications Objects** are specific to each application. They may use any *CORBA* objects (e.g. Object Services and Common Facilities). They are not standardized by *OMG*.

2.2. Obtaining an object reference

In order to invoke a method on a server object, a client must first hold an **object reference** to this server. An object reference is associated to at most one CORBA object. With an object reference, the *ORB* is in charge

of locating the object and delivering the invocation to the server.

Object references may be obtained by the following means. First, with `resolve_initial_references` *ORB* operation, clients may obtain references to well known services such as `InterfaceRepository` and `NameService`. Clients may also obtain object references from output parameters of any method. The Naming Service allows clients to obtain object references thanks to object symbolic names. And finally, the Trading Service allows clients to get object references selected thanks to a type of service name and a list of properties.

2.3. Trading Service

In this subsection we describe the main features of the trading service.

The trading service has been designed to allow the registration and discovery of objects. A **trader** is an object that provides the trading service in a distributed environment. Server objects advertise or **export** their service offers to traders. **Exporters** may be server objects or other objects acting on the behalf of the server. Client objects invoke traders to discover or **import** service offers matching a given type of service and a set of **properties**. Clients are called **importers**; they can be the consumers of the service or act on behalf of other objects.

A **service type** is defined in a Service Repository with an interface type and a set of zero or more properties. Each property is described by a name, a mode and a type of value. If a property mode is mandatory, then each instance of the service type must provide an appropriate value for this property when exporting its service offer. Each service type is identified by a unique `ServiceTypeName`.

A **service offer** consists of a `ServiceTypeName`, a list of properties (property name and value), and an object reference to the interface providing the service. Some properties are dynamic; for these, values are not in the service offer, but obtained explicitly from the interface of a dynamic property evaluator given by the exporter of the service.

The main trader interfaces are shown in Figure 2.1. The most important ones are the `Register` and the `Lookup` interfaces. The `Register` interface provides the `export` method for exporting a service offer. The `Lookup` Interface provides the `query` method for importing a list of service offers. The importer may express its **preferences** with a constraint language. Traders organize discovered service offers according to these preferences.

Traders can be linked together in a **trader federation**. Figure 2.1. gives a simple example of federation between traders *A*, *B*, and *C*. Target trader (e.g. trader *B*) establishes a link with a source trader (e.g. trader *A*) using the `Link` interface. Then, the source trader (i.e. trader A) is able to invoke target trader (i.e. trader B) with a `query` method. These links are therefore explicitly created and are unidirectional. All the links form a directed graph called the **trading graph**. The `Link` interface provides methods to manage the links, such as the `add_link` method used by a target trader to define a new link to a source trader, and the `remove_link` method to remove a link.

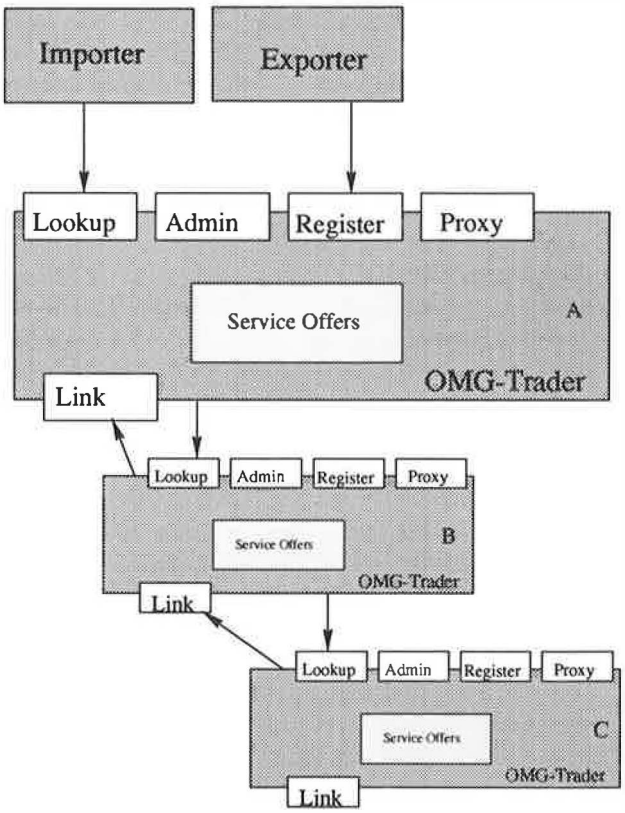


Figure 2.1: Traders Interfaces and Federation

A federation allows traders to extend an importation to a trading graph. **Importation policies** modify trader behavior for a discovery in a federation. Importation

policies are associated to each trader, each link, and each importation. A combination of these policies conditions the list of traders visited for an importation. Trader policy overrides link policy, which itself overrides importer policy. The main importation policies are given as follows: *(i)* `search_card`, gives the number of service offers to be searched; *(ii)* `return_card`, gives the number of ordered service offers to be returned to the client; *(iii)* `hop_count`, gives the maximum number of links that may be visited for a search; *(iv)* `starting_trader`, gives a path to a remote trader on which the search must start; *(v)* `follow_policy`, defines the trader behavior for propagating a search. The following policies are: *(i)* `local_only`, only locally registered service offers are returned; *(ii)* `if_no_local`, the search is only propagated if the number of local offers matching the request is less than the number of offers to be returned (i.e. `return_card`); *(iii)* `always`, the search is propagated till the expected number of offers is reached (i.e. `search_card`).

Besides the `Register`, `Lookup` and `Link` interfaces, the trading service defines three other interfaces. The `Admin` interface allows one to modify and list interfaces and policies supported by a trader. The `Proxy` interface is used to register service offers for which the object reference is not known at the exportation but obtained at query time thanks to a proxy object. And the `ServiceTypeRepository` interface is used for the management of the repository service types. Traders have to implement at least the `Lookup` interface.

2.4. Trading service optimization

In this sub-section, we present possible optimizations of *OMG* trading service and especially for that particular part which concerns trader federation.

Because of the huge number of services available on WANs, it is easier for end users to search a service according to its characteristics rather than to its name. Indeed, it's easier to get information on Internet through a search engine than by giving its URLs. Because of the increase in the number of services, the trading service is bound to become more useful than the naming service.

Trader federation is an interesting feature in the context of WANs: it allows one to distribute the trading service on several traders. However, improvement could be achieved on the following points.

As they are now defined, trader federations are bound to be established "manually" by an administrator. As a result, they may not be adapted to the underlying network topology. Furthermore, trader graphs may contain cycles (i.e. a search may visit the same trader several times).

As the federation is usually static, it cannot react in a transparent way to network events such as trader or communication link failures, or changes on the underlying network topology. Therefore, it doesn't adapt to events occurring on the underlying WAN.

The trading service doesn't define the concept of distance between objects. Consequently, the client cannot express the search of the nearest service, and the trader cannot organize service offers according to distance between clients and servers, even though this information could be important because of differences in communication costs in a WAN.

The integration of the *Cooperating Server Graph* model, which we describe in Section 3, in the trading service bring solutions to the above remarks and therefore would improve and optimize this service. We present in Section 4 a proposal of an *Extended Trading Service* using this model.

3. Cooperating Server Graph model

In this section, we summarize the *Cooperating Server Graph* model (**CSG**), which is described in details in [Taco97b]. The aim of the CSG model is to optimize and dynamically manage links between cooperating servers over a WAN. This model defines a protocol for dynamically updating the links according to different events happening, either to some servers, or to the underlying WAN. This model has already been adapted for the cooperation of WAN location servers for the *Chorus* micro-kernel [Taco97a]. We present in Section 4 the use of this model for dynamically managing a trader federation.

3.1. Cooperating Server Graph definition

On a WAN, we consider a set of **cooperating servers** (several hundred or so) which cooperate in order to

offer a service (e.g. a federation of traders which offers the trading service). The cooperating servers are geographically distributed on different computer **sites** (e.g. LANs) which may be separated by long distances. Each site is made up of physically close machines. The sites are logically linked (e.g. they belong to the same company or cooperate for a given project) and physically connected by an underlying WAN.

As there may be several sets of cooperating servers on the same WAN, we associate to each one a unique identifier (e.g. symbolic name).

The model uses a **distance** function. At a given time, this function associates a value to each couple of cooperating servers. This value has to be representative of the communication cost between the couple of cooperating servers (e.g. financial cost, latency induced by physical distance or available bandwidth), and may change over time. For Internet, we have used the distance function used by routing protocols (i.e. number of hops between sites).

We build a graph, namely a CSG, in which the nodes are the cooperating servers. The nodes are linked by a weighted edge providing that a distance value has been evaluated between the two nodes. The CSG is simple, k-connected and not oriented.

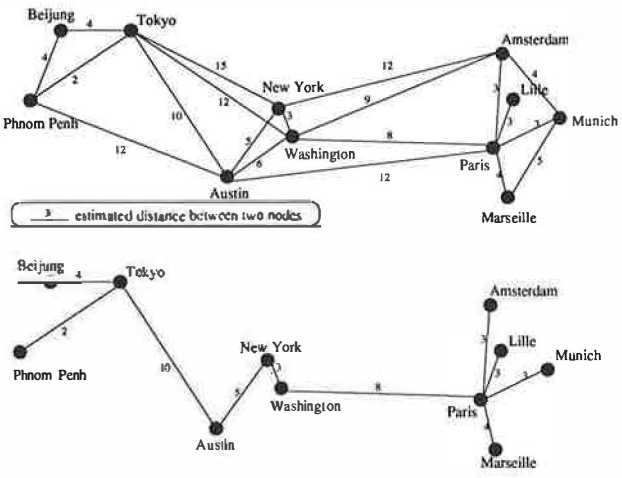


Figure 3.1: A CSG example (a) and its associated broadcast tree (b)

We assume that the number of nodes in a CSG is low (some hundreds or so). So each cooperating server stores in its own memory the layout of the CSG. Thanks to the CSG, all the cooperating servers calculate the same **broadcast tree** between all the nodes. We use a minimum-weight spanning tree calculated by the Prim algorithm [Prim57]. This tree is used to broadcast information to all the cooperating servers. With the broadcast tree, broadcasting is made with a minimum communication cost (according to the distance function), with a distribution of the communication load between all the nodes and without the need of a stop control for eliminating the CSG cycles.

Figure 3.1-a gives a simple example of a CSG with eleven cooperating servers distributed all over the world. Figure 3.1-b shows the broadcast tree calculated for this configuration.

3.2. Dynamic update of the CSG

The model includes a protocol for updating the CSG and its associated broadcast tree in order to take into account the following events: the addition or removal of a cooperating server, modification of the underlying WAN topology which leads to some CSG distance changes and temporary failure of a cooperating server or of a communication link.

In order to be more efficient and because of differences in the duration of events, the model offers three levels for taking those events into account: *(i)* the alternative behavior in case of failure; *(ii)* the local modifications; *(iii)* and the global change of CSG version.

When a failure is just discovered, i.e. when one node can't propagate an information to its neighbors in the tree, this node uses the alternative behavior in case of failure. It propagates the information on behalf of the failed neighbor[2] to the neighbors of the failed neighbor in the tree. For example, in Figure 3.1, if node *Phnom Penh* cannot propagate to node *Tokyo*, node *Phnom Penh* will decide to propagate to nodes *Beijung* and *Austin* on behalf of node *Tokyo*. This behavior is possible because of the global knowledge of the broadcast tree. This behavior maintains the continuity of the service.

The local modification level is used for long time failure, long time failure recovery, addition and removal

[2] *In order to simplify, we call it the failed neighbor but the communication failure may come from a failed server or from a network failure.*

of cooperating servers. A local modification consists in a coherent change of the broadcast tree seen on a node, its neighbors, and the neighbors of its neighbors. For example, if the failure of node *Tokyo* lasts after a given delay (e.g. several minutes), a new configuration of the tree shown on Figure 3.2 between nodes *Beijung*, *Phnom Penh* and *Austin* is calculated. This modification concerning node *Tokyo* is made coherently on *Tokyo*'s neighbors in the broadcast tree (i.e. *Beijung*, *Phnom Penh* and *Austin*) and on the neighbors of its neighbors broadcast tree (i.e. *New York*). Local modifications keep a broadcast tree, but this tree is no longer a minimum-weight spanning tree.

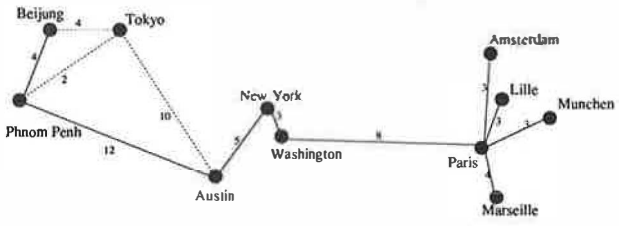


Figure 3.2: Local modification of the broadcast tree

A CSG version change is activated as soon as the degradation rate[3] of the broadcast tree goes past a given threshold. The activation is triggered by a Changing Version Server (**CVS**) chosen dynamically in the set of cooperating servers. The new version takes into account all the events considered as permanent since the last version: very long time failures, very long time distance changes, addition and removal of nodes. The new version is propagated on the broadcast tree. Two nodes have to agree on a version before they can communicate.

All the CSG updates are made dynamically. Thanks to the three update levels, the number of version changes is reduced. The links between all the nodes follow the evolutions of a CSG and its underlying WAN topology. The model tolerates a great number of failures. In case of too many failures leading to dividing the CSG into several isolated classes[4], server cooperation is limited inside each class.

We present in Section 4 how the integration of this general model optimizes a *CORBA* Trader federation.

4. The Extended Trading Service

4.1. Global description

The Extended Trading Service is an evolution of the OMG trading service, which integrates the Cooperating Server Graph model (cf. Section 3). The aim of this evolution is to optimize the management of trader federations and object importation over a set of traders scattered on a WAN.

The Extended Trading Service is offered by a federation of CSG-traders belonging to the same logical domain. A CSG-trader is a specialization of an OMG-trader preserving OMG-trader interfaces. For an importer or an exporter, the CSG-trader is therefore entirely conform to the OMG specification and offers the same service as the OMG-trader. Any implementation of the OMG trading service may be specialized in a CSG-trader.

We consider a CSG in which each node is a CSG-trader. Besides its OMG trading service function, a CSG-trader ensures automatic federation of CSG-traders. Each CSG-trader stores its `CSG` and calculates the broadcast tree. In a CSG-traders' federation, the propagation of importations follow the broadcast tree. Every CSG-trader manages dynamically (in collaboration with the other CSG-traders) the links of the federation. Links evolve according to events occurring on the network. CSG-traders may be linked to OMG-traders using OMG links. Object search is then performed according to client choice either in OMG mode using OMG links or in CSG mode using the broadcast tree. Finally, in order to reduce the number of extended importation, each CSG-trader manages a cache of service offers.

4.2. The CSG-trader architecture

The general architecture of a CSG-trader is presented in Figure 4.1. A CSG-trader consists of one OMG-

[3] *The degradation rate is estimated with the sum of the weights of the degraded broadcast tree and the sum of the weights of the minimum spanning tree that could be used.*

[4] *If a node cannot communicate with a node and some of the neighbors of the failed neighbor it assumes there is a partition. Its class is made up with the sub-trees with which the communication is still possible.*

trader part, called the trader part, and a CSG specialization called the CSG part.

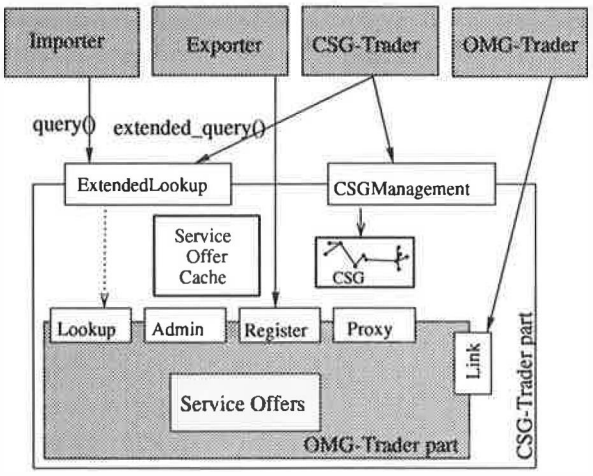


Figure 4.1: CSG-trader Architecture

4.2.1. The OMG-trader part

As a CSG-trader is derived from an OMG-trader, it inherits from all the OMG-trader interfaces namely the `Lookup` interface and possibly the `Register`, `Admin`, `Proxy` and `Link` interfaces. The `Link` interface is only used to create links between OMG-traders and CSG-traders, which we call OMG links. These links and their follow policies are managed by the trader part. Because links between CSG-traders have a different semantic they are managed by a special interface.

4.2.2. The CSG-trader federation

A CSG-trader federation is established according to the CSG. The links between CSG-traders, which we call CSG links, are entirely managed by the CSG part and are not explicitly created. Indeed, every CSG-trader knows all others CSG-traders, calculates a broadcast tree, and consequently knows its neighbors in the broadcast tree. A CSG link is essentially composed of the target CSG-trader name as well as a reference to its interfaces.

4.2.3. The CSG-trader specific interfaces

The extended trading service inherits interfaces from the *OMG* trading service. We add two new interfaces: the `CSGManagement` interface and the `ExtendedLookup` interface.

The `CSGManagement` interface provides methods for the management of a CSG. Among them we can mention `ask_for_csg` that allows a new CSG-trader to get the `CSG`, in order to set up a link with the nearest CSG-trader using the `add_extended_link` method.

The `ExtendedLookup` interface is a derivation of the OMG-trader `Lookup` interface. It provides several methods: *(i)* the overrided `query` for all clients (except CSG-traders); *(ii)* the `extended_query` for the propagation of a search over a CSG-trader federation; *(iii)* the `extended_answer` to return the result of a search to the source trader. Compared to the `query` method, the `extended_query` add two parameters: a `CSG` identifier and the name of the CSG-trader initiator of the importation. It is invoked in a one way method. The service offer result (if any), as well as the reference of the CSG-trader that gives the offer, is returned directly to the initiator CSG-trader later, using the `extended_answer` one way method. The initiator trader is then able to evaluate the distances of each discovered offer.

4.2.4. Service importation in a CSG-trader federation

In the CSG mode, a search is limited in a CSG. In order to allow clients to specify a `CSG` identifier, we define the `CSG` property. The `CSG` property is useful only for the CSG part of a CSG-trader.

When a CSG-trader receives a query request with its `CSG` identifier, it firstly asks its trader part with `local_only` policy (without the `CSG` property), and then, if necessary, propagates the request. If the CSG-trader does not belong to the indicated `CSG`, it can be considered as a **relay trader** and forward the request to a CSG-trader belonging to the required `CSG`. If the client doesn't indicate any `CSG` identifier, then the search is not carried out over the CSG federation but is accomplished following the OMG links exclusively.

4.2.5. Service offer cache

The goal of the CSG-trader is to optimize importation over a WAN. For this purpose, each CSG-trader stores a service offer cache in which it stores results of previous extended importations. All clients of the same CSG-trader benefit from this cache.

The service offer cache holds an LRU table in which each cell consists of a service type name, a set of properties, the policy used to discover this offer, and the reference of the CSG-trader that returned this service

offer. When a CSG-trader receives a `query` request it looks in the cache for a cell matching the `query`. If it finds any, it sends a "`local_only`" `query` request to the CSG-trader referenced in that cell in order to verify the validity of the service offer and get its dynamic properties. If the target CSG-trader returns the service offer, the cell age is updated; otherwise the cell is deleted.

4.2.6. Importation policies

The CSG-trader provides the same importation policies as the OMG-trader. We present here the importation policies whose semantic has been adapted to CSG federations.

With the `if_no_local` and `always` policies, the client request is propagated following the broadcast tree. Each intermediate CSG-trader invokes the one way `extended_query` method in parallel to all the following sub-trees. Results, if any, are returned directly to the initiator CSG-trader. With this behavior, the number of intermediate traders waiting for answers is significantly reduced, the drawback is that discovery goes on on each subtree independantly even if results have been found on other subtrees.

The `hop_count` policy preserves its semantic and applies to the broadcast tree. For example, with a `hop_count` of "1", only the initiator CSG-trader's neighbors on the tree will be visited.

Only the `if_no_local` policy uses the cache. In order to ensure the locality of the service offers, the cache is not used with the `local_only` policy. We also have chosen not to use the cache with the `always` policy, in order to preserve the quality of the results rather than the performance of the search.

Finally, if every server object exports its service offer to the nearest CSG-trader, and the client imports from the nearest one, the extended trading service may organize the results from the nearest to the furthest, thanks to the use of the CSG graph.

5. CSG-trader implementation

In this section, we first present the representation of a `CSG` in a CSG-trader, we then describe our CSG-trader prototype, finally we compare the OMG trading service and the Extended Trading Service with a simple federation example,.

5.1. Representation of a CSG

A CSG-trader federation is represented on each CSG-trader with the same data structure, in which each node or CSG-trader is described by a `TraderElement` which consists of the following components.

- The CSG-trader identification in the CSG.
- The distance function type (adapted to the federation).
- The reference of its `GSCManagement` interface (for dynamic CSG evolution).
- The reference of its `ExtendedLookup` interface (for query operation propagation).
- Its network address (for evaluation of distances between CSG-traders).
- The sequence of its neighbors in the broadcast tree (this information represents the broadcast tree).

A CSG is represented by a *CORBA* object (type `CSG`). In this object is stored: *(i)* the running version of the CSG (global information common to all nodes), and *(ii)* node specific information for recording local modifications. The different components of this object are as follows.

- `CSG` identifier
- The running CSG version number
- The number of CSG-traders (known locally). Because of local modification unknown on this node, this number may be different from the number of CSG-traders in the current broadcast tree.
- A sequence of `TraderElement`. This sequence may not be the same on each node because of local modifications.
- The distance matrix (distance between CSG-traders). If a distance has not been evaluated, the infinite value is attributed.
- The table of distance between the local CSG-trader and other CSG-traders. If the distance is infinite in the matrix, the distance is evaluated by the sum of distances in the shortest path between the two nodes (the CSG is connected).

A `CSG` is described by an *IDL* interface and may be obtained by another CSG-trader. This feature is interesting for the addition of a new CSG-trader in the CSG, and for the propagation of a `query` request to another CSG.

5.2. Prototype

We have implemented a prototype of CSG-trader on *Orbix 2.1*[5] with *Sun Solaris 2.5*. This prototype uses *IIOP* references.

At the time of our implementation we did not have the sources of any OMG-Trader, so we have implemented a **simplified OMG-trader** which supports `Lookup`, `Register` and `Link` interfaces. But we can easily adapt our prototype to any OMG-trader implementation.

Our CSG-trader prototype is a specialization of the *simplified OMG-Trader*, which furthermore implements `CSGManagement` and `ExtendedLookup` interfaces, provides dynamic updates of the CSG and manages the *offer service cache*.

With this prototype we have done the following elementary tests on a LAN.

- Addition and removal of a CSG-trader in the federation.
- Automatic CSG version change on all the nodes.
- Propagation of importation on a CSG-trader federation with `local_only`, `if_no_local` and `always` policies.
- The alternative behavior in case of failure of an intermediate CSG-trader.
- CSG-trader and OMG-Trader cohabitation.

5.3. Comparison between a CSG-trader and an OMG-Trader

In order to give an interesting comparison between a CSG-trader and an OMG-Trader we would have needed a complete *OMG-trader* implementation and a testbed for WANs. We did not have any of these two conditions, that is the reason why we don't give any performance comparison. We present here a comparison illustrated by an example in order to highlight interesting features of the Extended Trading Service.

In this section, we use the CSG federation of Figure 3.1. An example of possible associated *OMG* trading graph is given in Figure 5.1-a. In Figure 5.1 unidirectional OMG links are represented with one arrow, other links are bi-directional. In this figure we present the invocations needed for a discovery in the OMG graph (Figure 5.1-a) and in the CSG federation (Figure 5.1-b).

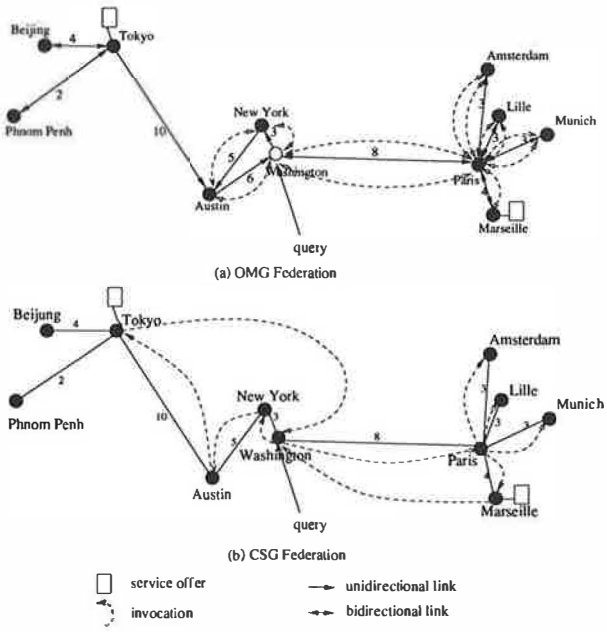


Figure 5.1: Example of discovery

For this example, we compare the number of method invocations needed for OMG Trading Service and Extended Trading Service. The query request example uses the policy `if_no_local`, the `search_card` and the `return_card` both have value 1, the initiator trader is *Washington*. The matching service offers are on nodes *Tokyo* and *Marseille*. The Extended Trading request propagates requests in parallel on all the following sub-trees. Dashed arrows symbolize method invocations.

For this example, an importation on OMG federation generates 12 two-way method invocations and on the CSG-federation it generates only 10 one way method invocations. In OMG federations, double links (bi-directional links) lead to double invocations (even though there is a stop control). So, even if the OMG-trader graph is a tree, two times more invocations would be needed. Number of invocations will also be reduced by the CSG trader cache management.

In order to avoid cycles, the OMG-traders need to store and compare request identifiers (case of cycle *Washington*, *New York* and *Austin*) at each node.

[5] Iona Technology *CORBA implementation*

With an OMG-trader federation, no guarantee is given on the existence of a path between each pair of nodes. In the example, the *Tokyo* service offer can't be found. Special attention is needed to configure an OMG-trader federation.

With an OMG trader federation each intermediate trader in the importation has to be waiting for an answer (RPC invocation). With the CSG federation only the initiator trader is waiting for an answer.

We argue that CSG-traders would facilitate trader federation. However, more tests are needed to verify the efficiency of the overall CSG federation mechanisms.

6. Conclusion

Because of distributed computing, the evolution and diversity of services offered on today's and tomorrow's WANs, discovering tools such as the trading service should become essential for end users.

In this paper, we have described the *CORBA* trading service specified by the *OMG*. With this service clients may import service offers exported on traders. Cooperating traders may be federated to offer extended searches. Yet, we have shown that as links are statically and manually established they are not adapted to the underlying network topology and do not evolve dynamically. Moreover, they do not help clients to choose the nearest replicated object, while, because of communication delay and cost on WAN; this would be an important feature.

We have presented the *Cooperating Server Graph* model. With this model, the links between cooperating servers are established dynamically. Furthermore, thanks to an inter server protocol, the links evolve to react to different events such as intermediate servers or communication links failures, and modifications in the underlying network topology. With this model, the propagation of information to all servers is efficient. The knowledge of distance information between the servers allows traders to organize the results from the nearest to the furthest.

We have defined the CSG-trader that integrates the CSG model in an OMG-Trader. CSG-traders offers the following optimizations. Trader federation is established and evolves dynamically. Extended service offers searches follow a minimum-weight spanning tree. Assuming that importation and exportation are sent to the nearest trader to clients and servers, searches may find the nearest server to each client. Asynchronous treatment of requests increases the number of requests handled in parallel by each trader.

We have then described a CSG-trader prototype for *Orbix* 2.1, which is a specialization of a simplified trader that we have implemented.

In the definition and implementation of the CSG-trader we paid a special attention to stay conform to the trading service interface specification. Our optimizations are transparent for trading service clients. In order to facilitate the choice of a search domain (i.e. a CSG), and of a starting trader, it would be interesting to adapt the trader specification.

Trading service and migration

We would like to emphasize that migration, taken into account in *CORBA* life cycle service, and trading exportation should be linked together. In *CORBA* specification, an object reference should stay valid after a migration. So, a service offer stays valid after a migration. Yet, in order to both facilitate *ORB* location service and preserve the nearest server semantic offered by CSG-traders in case of object migration, we argue that a server migration should be coupled with the migration of its associated service offer. And so service offers will be registered on the trader which is the nearest to the server object.

CORBA domains

CORBA specification defines several notion of domains (interoperability domains, policy domains, security domains). A more precise definition of administrative domain seems to be an important issue.

Just like CSG, a domain may take into account the logical relationship between *ORBs*. For example, all the computer sites of a company may define a *CORBA* domain. Some of the *CORBA* services could benefit from such domain definition. The life cycle service could limit some migration and replication inside a *CORBA* domain. The trading service could restrict searches inside a *CORBA* domain. Every object would be associated to one or several administration domains. And so, some operations may be authorized

between objects of the same domain only, while others may be authorized between different domains.

References

[Bern94] T. Berners-Lee, L. Masinter, M. McCahill. Uniform Resource Locators (URL), *RFC1738*. December 1994.

[Bris95] T. Brisco. DNS Support for Load Balancing. RFC 1794, April 1995.

[Deer90] E.S. Deering and D.R. Cheriton. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM transactions on Computer Systems*, 8(2), May 1990.

[Moat97] R. Moats. URN Syntax. *RFC 2141*, May 1997.

[Mock87] P. Mockapetris. Domain Names Implementation and Specification. *RFC1035*, November 1987.

[ODP93] Information Technology- Open Distributed Computing – ODP Trading Function. *ISO/IEC JTC1/SC21.59 Draft, ITU-TS-SG 7 Q16 rapport*, November 1997.

[OMG96] Trading Object Service. *OMG Document 96-05-06, RFP5 submission*, May 1996.

[OMG97] Common Object Request Broker: Architecture and Specification. Revision 2.1 *OMG Document*, August 1997.

[Prim57] R.C. Prim. Shortest Connection Networks and some Generalizations. *Bell Syst. Techno. J.* 36, 1957.

[Seit96] R. Seitzer, E.J. Ray, and D.S. Ray. Alta Vista Search Revolution: How to find anything on the Internet. *Digital Press*, New Jersey, 1996.

[Taco97a] C. Taconet and G. Bernard. Object Location in Wide Area Networks. In *Porceedings of ERSADS'97 European Research Seminar on Advances in Distributed Systems*, Zinal, Switzerland, March 1997.

[Taco97b] C. Taconet. Graphe de Réseaux Coopérants et Localisation Dynamique pour les Systèmes Répartis sur Réseaux Etendus. *Ph.D. Thesis* University of Evry, France, October 1997.

[VS96] M. Van Steen, F.J. Hauck, and A.S. Tanenbaum. A model for World wide Tracking of Distributed Objects. In *proceedings of TINA'96*, Heidelberg, Germany, September 1996.

Filterfresh: Hot Replication of Java RMI Server Objects

Arash Baratloo*

P. Emerald Chung
Sampath Rangarajan
Yennun Huang
Shalini Yajnik†

Department of Computer Science
New York University, 251 Mercer Street
New York, NY 10012

Lucent Technologies Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

Abstract

This paper presents the design and implementation of a Java package called *Filterfresh* for building replicated fault-tolerant servers. Maintaining the correctness and integrity of replicated servers is supported by a `GroupManager` object instantiated with each replica to form a logical group. The Group Managers use a *Group Membership* algorithm to maintain a consistent group view and a *Reliable Multicast* mechanism to communicate with other Group Managers. We then demonstrate how *Filterfresh* can be integrated into the Java RMI facilities. First we use the `GroupManager` class to construct a fault-tolerant RMI registry called *FT Registry*—a group of replicated RMI registry servers. Second, we describe our implementation of the *FT Unicast*—a client-side mechanism that tolerates and masks server failures below the stub layer, transparent to the client. We also present initial performance results, and discuss how general purpose RMI servers can be made highly available using the *Filterfresh* package.

1 Introduction

Distributed object technologies have become popular in developing distributed applications. Among these, object technologies such as CORBA [17], DCOM [4], and Java Remote Method Invocation (RMI) [22, 20] are the most popular. Although these middleware platforms ease the development of distributed applications, they do not directly improve the reliability of these applications. As a result, application developers have to implement their own mechanisms to improve the reliability and availability of their applications. The task of developing fault tolerance techniques for distributed object paradigms is often tedious and error-prone. Therefore, there is a great need to develop a generic, portable and reusable tool that enhances the reliability and availability of distributed objects.

In this paper we focus on using *Java RMI* to implement reliable objects. In Java RMI, an application consists of client and server objects. A client invokes a server's method using the server's object reference. To make its object reference available to clients, a server registers a tuple containing its object reference and a string name with a name-server called *RMI registry*. This operation is called *binding*. Given a string name, clients can get the remote reference of a server registered under that name by contacting the RMI registry. This operation is called the *lookup*. There could be many registries running in a network, but, registry data sets among different registries are not shared or replicated. Therefore, a client must have *a priori* knowledge of hosts running RMI registries. From the fault tolerance point of view, the current registry implementation is a single point of failure for RMI applications. For example, if one registry fails all of its data is lost and clients cannot get object references of servers running at

*baratloo@cs.nyu.edu. This work was done while the author was a summer intern at Bell Laboratories.
†emerald,yen,sampath,shalini@research.bell-labs.com.

that site anymore. As a result, even though the servers are still alive they may not be accessible. This problem becomes even more complicated when we make servers migratable which forces them to re-register after a migration. Therefore, to make it possible for clients to find servers running on remote hosts unknown to them, and to make RMI applications fault tolerant, it is necessary to enhance the registry mechanism.

One way of improving the registry mechanism is to replicate its data sets among all nodes. Clients can then query any node on a network to get a server's object reference without the need of identifying the right registry first. To achieve this, we adopted the hot replication scheme, i.e., updates to any node are reliably propagated to all other nodes, and changes are made *consistently* on all sites. This approach ensures that the registry data sets are strongly synchronized.

We use the virtual synchrony model [3] to implement *FT Registry*, our group of replicated RMI registry servers. Virtual synchrony and its underlying process group operations are provided by toolkits such as Isis [3] and Transis [6], in Java middle-ware systems such as iBus [15], and in operating systems such as Amoeba [11] for building fault-tolerant applications. Based on the success of such systems, same mechanisms are used in Orbix+ISIS [9] and Electra [13] for adding fault-tolerance to CORBA, in the work proposed by [1] for adding fault-tolerance to other Object-Oriented systems, and in systems such as [12, 14] in providing fault-tolerant distributed Name Servers. Our challenge here is to integrate such mechanisms into the Java RMI system with minimal changes while staying 100% Pure Java.

Our *FT Registry*, in addition to being able to tolerate failures itself, provides the building block for fault-tolerant RMI application servers. For example, a crashed server can be restarted on a different node and it can register with another *FT Registry*. Since the server's reincarnation (i.e., the new registration) is propagated to all nodes, any client on the network can lookup the server's new object reference. This does not solve all the problems however. In the mean time, clients holding the old object reference may invoke remote operations which will fail. To recover from such a failure, we provide *FT Unicast*. The *FT Unicast* object works below the stub layer and gets a valid object reference and retries the invocation whenever a server failure is detected, thus making the server migration and fail-over transparent to client applications.

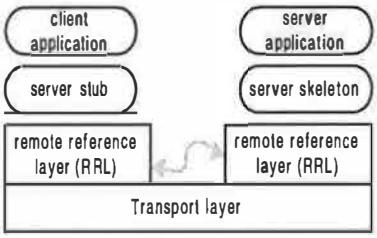


Figure 1: RMI architecture

In the next Section, we describe the Java RMI architecture. Section 3 provides an overview of the *FT Registry* and *FT Unicast* fault-tolerance mechanisms. Section 4 describes the `GroupManager` class that is used to manage the group of replicated *RMI registries*. Sections 5 and 6 describe implementation details of the two fault-tolerance mechanisms and give initial performance results. Section 7 describes how the `GroupManager` can be used to provide fault-tolerance to general Java application servers through replication. Conclusions are presented in Section 8.

2 RMI Architecture

We briefly describe the Java RMI architecture in SUN's JDK1.1 reference implementations. In a nutshell, *Java RMI* enables an object (client) to invoke methods of certain interfaces implemented by another object (server) running on a different *Java Virtual Machine* either on the same host or on a different host.

The RMI architecture consists of three layers as shown in Figure 1: the *stub/skeleton* layer, the *remote reference* layer (RRL) and the *transport* layer [22]. On the server side,

for an interface to be invoked remotely, it has to be derived from the `Remote` class. The object that implements this interface may derive from the `UnicastRemoteObject` class of the RMI package. The current `UnicastRemoteObject` uses *TCP* for low-level transport.

The *rmic* complier takes a server object implementation and generates two class definitions, a stub object for the client and a skeleton object for the server.

On the server side, when the server object is created, the constructor of `UnicastRemoteObject` performs an `exportObject()`. Inside `exportObject()`, an `UnicastServerRef` object is instantiated and exported. It creates a live reference object (the transport layer) which contains an *IP address*, a *TCP port* number and an *Object ID*. It also creates the skeleton and the stub at the server side. Then a mapping from the Object ID to the stub and skeleton is registered in an *object table* residing in the transport layer.

On the client side, the application obtains a reference to the server object from RMI registry or from other objects. If the client does not have the stub code in the local host, the stub is dynamically loaded from the server side. The stub is a layer between the application and the lower layers of the RMI mechanism. The main function of the stub is the marshaling/unmarshaling of requests and results and passing them between the client and the Remote Reference Layer. The client stub contains a `RemoteRef` object. The `RemoteRef` object encapsulates the transport layer underneath. The transport layer gets the live reference of the server object and establishes the connection to the server side. The client stub calls `invoke()` method in `RemoteRef` to make the call to the remote site. Once the call gets to the server side endpoint, the server side transport checks the object table and maps the Object ID to the corresponding skeleton to dispatch the request. The skeleton unmarshals the parameters from the request and then makes the up-call to the object. The results are marshaled by the skeleton and passed back to the client side.

RMI registry is a simple name server provided by the RMI package. A server object registers a name using the `bind()` method call. The registry keeps a name to remote object mapping. It listens at a well-known port, typically, `1099`. Any client can get a reference of a remote object by name via the `lookup()` method call.

3 Overview of *FT Registry* and *FT Unicast* Fault-Tolerance Mechanisms

Filterfresh is a Java package for building highly-available servers in presence of processes crashes and network failures. In applying *Filterfresh* to Java RMI, we have implemented a Fault-Tolerant Registry (*FT Registry*) service. This service is then used to mask server failures in RMI client/server applications at the client side, completely transparent to the client (*FT Unicast*).

3.1 Replicated RMI Registry - *FT Registry*

RMI registry with the "local registry" requirement where application servers can `bind` services only with the registry local to the server machine, is too restrictive for failure recovery. This also restricts the dynamic migration of servers from one machine to another since there is no standard method for clients to find the location of application servers. We can eliminate the problem of the registry being a single point of failure and the problem of locating application servers by replicating the registry and distributing the replicas over different machines on the network. Thus, we provide a replicated RMI registry on a network, and manage the replicas to maintain consistent data sets. We also perform failure detection of the RMI registry replicas and if the registry replicas are manually restarted, enable them to transfer state from one of the available replicas and synchronize their state.

The main problem then is to keep all replicas of the registry servers synchronized in

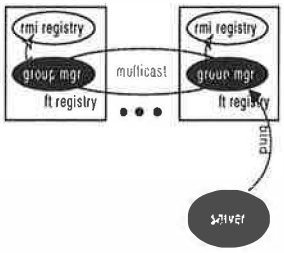


Figure 2: Server binds with the *FT Registry*

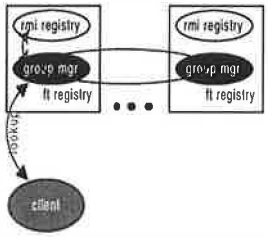


Figure 3: Client looks-up the *FT Registry*

spite of process failures and network failures. It is well known that the *process group approach* tolerates these failures. A solution based on the process group approach would provide the following.

1. Allow the replicas of the registry server to form a group.

2. Let each of the replicas maintain a consistent view of the group; i.e, let them be aware of who is in the group and who is not, in a consistent way.

3. Let the replicas propagate updates through a group multicast primitive (this is performed through a `GroupManager` class explained below); for example, if a server object `binds` with one of the registry replicas, this will be reliably propagated to other replicas so that they can update their data set to reflect this event.

4. Provide for total order on the messages that are used to propagate updates so that data sets are updated, by all replicas, in the same global sequence and hence in a consistent way; for example, if a server *A binds* with one of the registry replicas while another registry *B joins* the group of registry objects, we guarantee that the two events will be observed in the same order by all replicas. This ensures that either (1) the data set transferred to *B* is the image before *A*'s registration followed by the registration event, or (2) the data set transferred to *B* is the image after *A*'s registration.

The server group is managed by implementing a *Group Membership* algorithm. We provide a Java `GroupManager` class that implements the group membership algorithm using a *Reliable Multicast* primitive. The `GroupManager` object is instantiated in each replica of the *RMI registry* server as shown in Figure 2. The group managers used by the different replicas of the RMI registry form a process group. The `GroupManager` object supports the following operations.

1. *Group creation:* When a registry server is instantiated for the first time, its group manager creates a group with this as the only member.

2. *Join:* When another server replica is instantiated, its group manager *joins* the group by first transferring state from an existing replica, and then updating the group view (before any other operation can take place). This ensures uniform view and consistent states among replicated registries.

3. *Leave:* A server replica is allowed to *leave* the group.

4. *Failure detection:* The group managers *ping* other group managers periodically and if they detect a failure perform a change of view for the group.

5. *Reliable multicast:* Guarantee that messages directed to all group members are atomic and totally ordered across all replicas. The group managers themselves multicast the above group operations (such as *join*), and *FT Registry* servers multicast registry operations (such as `bind`) using the reliable multicast provided by the `GroupManager` class.

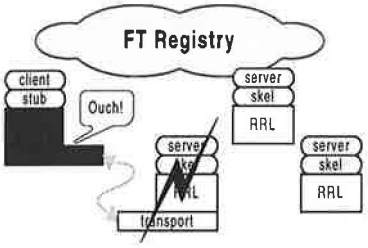


Figure 4: Application server failure detected on a remote method call

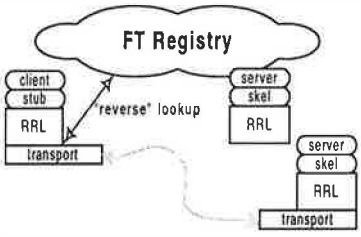


Figure 5: Stub does a reverse lookup on the *FT Registry*

Figures 2 and 3 show examples of how the `GroupManager` object is used by the *FT registry*. We adopt the *write-all-read-one* semantics. In Figure 2 when an application server binds with the local RMI registry, this information is updated locally as well as reliably multicast to all other group managers so that they can update their RMI Registry replicas. Figure 3 shows a client performing a `lookup` operation. In this case, the lookup is performed locally and is not multicast to the other group managers.

The `GroupManager` implementation is described in Section 4 and the *FT Registry* implementation using the `GroupManager` is described in Section 5.

3.2 Transparent Client-Side Fault-Tolerance - *FT Unicast*

FT Registry allows multiple application servers providing the same service and running on different hosts to register under the same name. Thus, RMI clients observe the same interface using our fault-tolerant services as with standard RMI servers, however, we use this feature to mask server failures completely transparent to clients.

We accomplish masking of server failures as follows. In a standard client/server RMI applications, a client first gets a remote reference of a server object, typically by a name lookup from the registry server. When the client makes a method call on a non-faulty application server, the stub uses this remote reference to contact the server. On the other hand if the application server has failed, an exception is raised. Now consider a set of replicated application servers registered with a group of *FT Registry* servers under the same name. If the application server has failed, the raised exception is caught at the remote reference layer as shown in Figure 4. The remote reference layer performs a `reverse lookup` at any of the registries using the *stale* reference to the faulty application server, as shown in Figure 5. The *FT registry* returns the *name* of the faulty application server. This name is used to make a normal `lookup` to get a fresh reference to an available replica of the server object. The method invocation is retried with the new server and the results are returned to the client. This provides an illusion of a valid object reference to the client. The client is unaware of the actions that the remote reference layer takes between the time it makes a remote method invocation to the time it receives the results.

The *FT Unicast* implementation is explained in more detail in Section 6. In the next section we describe the `GroupManager` class implementation.

4 Implementation of the `GroupManager`

The process group approach is at the heart of our system in providing fault tolerant services. The process group functionality is provided by the java `GroupManager`

class. To achieve fault-tolerance, any object, such as the RMI Registry, can instantiate a `GroupManager` object and use its services. In the rest of the section, we will refer to objects that instantiate and use the services of the `GroupManager` as *clients*. The `GroupManager` class ensures *atomic* and *totally ordered* group operations in presence of *crash* failures. The atomicity assures that that an event is either seen by all group members or none. The total ordering assures that all group members observe the events in the same relative order.

The protocol assumes an *unreliable* point-to-point message delivery. The communication is implemented with *UDP* [19]. *UDP* datagrams are unreliable, and hence, appropriate mechanisms such as acknowledgement, retries, and timeouts are provided at a higher level to ensure correct group operations. We chose UDP as opposed to other protocols, such as TCP, for three reasons. First, a connection-less protocol is less rigid and can tolerate transient network outages. Second, since our system had to incorporate appropriate high-level mechanisms for communication and processes failures, any buffering and retransmission by the communication layer would have been redundant. And finally because UDP is faster. In retrospect, and as the experiments will show, the performance gained by using UDP did not have a large impact on the overall system performance.

A `GroupManager` object runs its own thread of control. Client-to-`GroupManager` interactions such as multicast, are done through method invocations. On the other hand, `GroupManager`-to-client interactions, such as `GroupManager` informing a client application of receipt of a multicast message, are done through asynchronous call-back functions. Through our initial experiments with building fault-tolerant systems such as the *FT Registry*, we have found that call-backs work well in integrating group membership services into object oriented systems. We are considering other models for future implementation, in particular, the new event model introduced in Java version 1.1.

4.1 Group Operations

The `GroupManager` class implements the following five basic operations: group creation, join, leave, reliable multicast, and reset group view. We describe each operation in turn.

Group Creation: A `GroupManager` object can create a new group at any time by invoking the public method `createNewGroup()`. This invocation results in creation of a new group having the `GroupManager` object as its only member. Once it has become a group member, the `GroupManager` object can be queried for other group members, the leader of the group (described below), and it can multicast messages to all members.

Join: A `GroupManager` object that does not already belong to a group can join an existing group. The public method `joinExistingGroup()` takes a host name and a port number (of any one of the group members) as parameters. Once `joinExistingGroup()` is called, the control is passed to the `GroupManager` object and the calling thread blocks. The `GroupManager` provides the atomicity and the total ordering of the join operation by using the group *reliable multicast* operation (as described below). Once the original group members receive the join event, the state of one of the original members is transferred to the joining member. In our implementation we have found that object serialization is a convenient mechanism to implement state-transfer. After the state of the new member is brought up to date, the calling thread is unblocked.

Leave: The leave operation is implemented by the public method `leave()`. Its implementation is analogous to the join operation in blocking the calling thread, multicasting the leave event, and unblocking the calling thread when the multicast succeeds.

Reliable Multicast: In every process group there is a distinguished member called

the *group leader*. The *group leader* runs the same code as other members, and interacts with the client application the same way, the only difference is that it has more responsibility. If the group leader crashes, or if another member suspects it of crashing, the group can elect any other member to function as the new leader.

When a client application invokes the `createNewGroup()` method of an `GroupManager` object, it results in the creation of a new group. The `GroupManager` object is the only member of this group, and by default, it becomes the group leader.

A `GroupManager` object exports a public method called `multicast()` that can be invoked to send an atomic and totally ordered multicast message to all the group members. When a client invokes the `multicast()`, the message is passed to the `GroupManager` thread and the calling thread blocks. The `GroupManager` stores a copy of the message in a local buffer, then forwards it to the group leader and waits for an acknowledgement of the operation's success before unblocking the calling thread. This message is not guaranteed to reach the group leader since UDP datagrams are unreliable. For this purpose, the `GroupManager` sets up a timer and resends the message to the group leader if the timer expires before receiving the acknowledgement. When the group leader receives the message, it increments a message sequence number and sends the message along with the sequence number to all group members. The sequence number serves to ensure duplicate messages are handled properly. Once the group leader receives the acknowledgements, it notifies the object that initiated the multicast that the operation has succeeded. On the other hand, if the group leader fails to receive the acknowledgements after a set number of retries and within a given timeout period, it initiates a *reset group view* operation to recover from potential failures.[1]

[1]In practice, we observed that system performance is very sensitive to the timeout period and the number of retries. If the numbers are set too high, the system takes a long time to detect failures or to resend dropped message. For numbers that are set too low, it causes the system to send excessive messages and to initiate failure recovery too often.

From the above discussion, we see that every group operation is issued from the same process, namely the group leader, and operations are carried out one at a time. Therefore, in the absent of process crashes, every group operation is atomic and group members observe the events in the same order.

The multicast protocol that we implemented can be categorized as *ack-based* since messages require explicit acknowledgement. See [2, 10, 16, 11] for other protocols that are not *ack-based* but provide the same semantics.

Reset Group View: Informally, a *group view* refers to the list of group members that a `GroupManager` object knows about, along with the *unique id* of each member, the identity of the group leader, and a *view incarnation* number. The view incarnation number is a counter that is incremented with each view change. The view incarnation number is included in every message and it serves to ensure that a message directed to an old group will not be accepted by a new group.

Reset group view refers to a member initiating *failure detection* and wanting to reestablish the group view. This operation is generally used to recover from failures, that is, after one `GroupManager` suspects another of failure. However, a client application can, at any time, initiate a reset group view operation by invoking the public method `resetView()`. Once a `GroupManager` object enters a reset view mode it blocks all other operations until a new view is installed.

Our reset view protocol is based on [11]. It runs in two phases. The first phase of the protocol determines a new group view, i.e. establishes which members are non-faulty and chooses the group leader; the second phase of the protocol brings the members up-to-date, and then installs the view determined in the first phase.

In the first phase, any `GroupManager` that invokes the `resetView()` method becomes a *coordinator*. Thus, there may be more than one coordinator at a given time running the first phase. A coordinator invites other

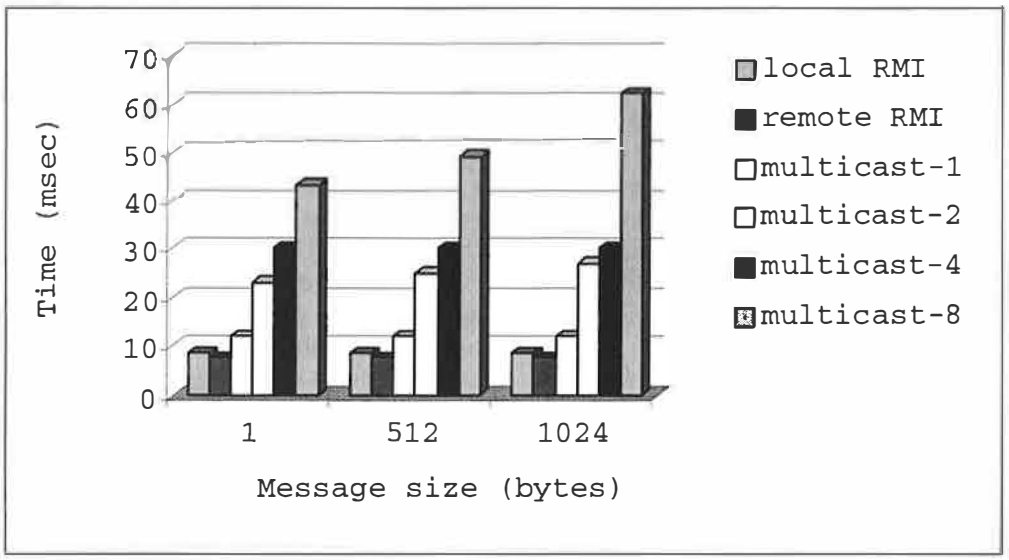


Figure 6: Performance of group multicast operation. The x-axis denotes the message size and the communication method. The y-axis represents time in milliseconds.

members to create a new view by sending a `request_view_change`. A non-faulty member that is not a coordinator accepts the invitation by responding with `ok_view_change` message. A coordinator accepts the invitation of another coordinator only if the inviting coordinator has a larger *id* number. Once a coordinator has received `ok_view_change` messages from a majority of group members, it continues to the second phase. If a coordinator is not able to successfully invite enough members within a timeout period, it repeats the first phase again. If a non-coordinator has not installed a new view within a timeout period, it becomes a coordinator and starts the first phase. Because it is required for a coordinator to successfully invite a majority of old group members, at most one coordinator could reach the second phase.

In the second phase, the coordinator first makes sure that every member has the latest message, i.e., the message with the largest sequence number. It then creates a view with the new members, the new group leader (itself), and the new incarnation number. The view is then sent to every member to install. The second phase completes when the coordinator receives an acknowledgement from every new member. If the coordinator does not receive all acknowledgements within a specified time, it repeats the first phase again.

Notice that the process of constructing a new group view will block until enough surviving group members can be found. It is well known that it is impossible to have a deterministic, correct and terminating algorithm to achieve consensus [7] in the presence of even a single failure and to build reliable failure detectors [5]. In the presence of these negative results, this protocol guarantees correctness *if and when* it terminates—that is, it will block until a consistent state can be constructed. Specifically, this protocol guarantees that if it terminates (1) all surviving members have a consistent group view, and (2) all the members in the new group view successfully receive all the messages sent by any member of the original group view before the failure.

4.2 Experiments

Here we present initial performance results for our reliable group multicast implementation. Experiments were conducted using up to 8 PentiumPro/200 machines connected by a Fast Ethernet hub. We used JDK1.1.1 running on Linux RedHat 4.0, and compiled with

optimization turned on. Reported times are *elapsed times*, and hence account for all overheads.

We measured the elapsed time for a group multicast operation to complete as measured from the invocation of the multicast until the invocation thread unblocked. This includes the time for the client to forward the message to the group leader, the group leader to reliably multicast the message, and then for the group leader to acknowledge the success of the multicast to the initiating client. We timed the operation for groups of size 1, 2, 4 and 8, and messages of size 1, 512 and 1024 bytes. With the exception of the group of size one, the multicast was initiated from an arbitrary member other than the group leader. We also measured the time for sending equivalent size messages using a single Java RMI. The results are shown in Figure 6.

Our first observation is a counter-intuitive one. We found that RMI is faster across two remote machine than on single host. We contribute this to the inefficiency of the Java runtime system we used—the faster intra-machine communication could not compensate for the shortage of resources. We were also surprised by the inefficiency of multicasts to groups of size one. When a group consists of only one object, there are message processing times, but there are no message transmissions. Multicasts took approximately 12 milliseconds. We attribute some of this to inefficient Java threads implementation under Linux. For example, we found that threads blocked on user inputs are never preempted—the work around seemed to have been expensive. Furthermore, we used object serialization for constructing low-level control messages, and as reported in [8], there is a high overhead associated with object serialization due to inefficient buffering and copying of the data. Considering that we sacrificed efficiency for simplicity in choosing the multicast algorithm, the `GroupManager` class shows reasonable scalability. For example, in increasing the group size from 1 to 8, we observed an average slowdown of 5.

5 Implementation of *FT Registry*

The `GroupManager` described in the last section is used to build the *FT Registry*. In Java RMI terminology, a *registry* is a remote object that provides a basic name server functionality. The `Registry` interface and the `LocateRegistry` classes provide this functionality. Two methods provided by the RMI registry are of special interest to us: `bind()` — to map a remote (server) object to a string (service name), and `lookup()` — to get a remote object associated with a string. The `rmiregistry` provided in JDK1.1 is a shell-script command that invokes `RegistryImpl`, an implementation of the `Registry` interface.

5.1 Replication Approach

A limiting factor of the existing RMI system is that the `RegistryImpl` successfully binds an object only if it is local to its machine. This introduces two problems in building client/server systems based on *Java RMI*. First, a client must have *a priori* knowledge of the host running the registry and the server. Second, the RMI registry becomes a single point of failure.

We address both problems by providing a replicated registry service, and by maintaining a consistent state among all replicas through the state machine approach [18]. Our implementation consists of the `FTRegistry` interface, and `LocateFTRegistry` and `FTRegistryImpl` classes. We also extends the standard interface by introducing the `multiBind()` method. The `multiBind()` method is a mechanism for multiple replicas to register under the same name. When this happens, the `lookup()` method returns an arbitrary object at random. This means that by replicating critical services, their loads will also be dispersed without client awareness, and without any effort on the part of the programmer.

Our `FTRegistryImpl` class is a replicated implementation of `FTRegistry`, replicated in the sense that instances of this class form and maintain a logical group for the du-

ration of their existence. By default, the first `FTRegistryImpl` object forms a singleton process group. Other replicas perform a group-join and a state-transfer, in which the state of the new member is brought up-to date, before becoming functional. The management and the communication among the group members are provided by embedded `GroupManager` objects.

A `FTRegistryImpl` object is composed of two logical layers: the *RMI registry* and *group manager* layers. The registry layer contains the actual data structures for object name-reference mappings. It is through private methods implemented at this layer that mappings can be added, removed and queried. The public methods such as `lookup()`, `bind()` and `multiBind()` are wrappers. When such methods are invoked, depending on whether the operation *alters the state of the registry map*, they are either passed up to the registry level to execute the corresponding private methods on the local data, or passed down to the group manager layers to multicast to all replicas.

To ensure consistent states across all `FTRegistryImpl` objects, operations that alter the registry map must be executed by all replicas. For illustration purposes, consider the case when a server object invokes the `bind()` operation of a `FTRegistryImpl` running on its local host. This is depicted in Figure 2. The group manager inspects the method and determines that the execution will result in modification to the registry map. Since the operation needs to be executed by every replica, the group manager sends the event to others using the `GroupManager`'s group multicast. This ensures that the maps of all registry replicas contain the same information at all times.

The hot replication of registry maps has two clear advantages. First, the RMI naming service will no longer remain a single point of failure. Second, it simplifies the `lookup()` operation. Clients no longer need *a priori* knowledge of the server's host, since a `lookup()` operation performed by any registry (see Figure 3) will return a server registered anywhere on the network.

5.2 Supporting `Reverse Lookup`

As mentioned earlier, we have implemented a system that can transparently mask server failures. Because of the *transparency* requirement, we had to work below the code that is generated by the Java and RMI compilers, see Figure 1. For this purpose, a fault has to be detected and masked at the `Remote Reference Layer` (*RRL*) or below. However, at the *RRL* level, concepts of server objects and server names do not exist, there are only remote reference objects and connections. Thus we need a mechanism that could construct a connection to a replicated server, given a *stale* connection to a crashed server.

We addressed this problem as follows. First we extend the mappings of our *FT Registry*. For each server object, in addition to storing its name and remote object, we store its connection object (live reference). With the added information a *reverse lookup* operation, where a server name can be looked up based on its live reference, becomes possible. Thus, given a server's live reference, our *FT Registry* can return references to other replicas of that server. In the next section we will discuss how this functionality is used.

Also note that registering the live reference is transparent to users—the server object simply calls `bind()` or `multiBind()` methods of our `DistributedNaming` class that extends the standard `Naming` class. Accessing the live reference and passing it to the `FTRegistryImpl` are hidden in our implementation.

5.3 Experiments

We measured the time for `bind()` and `lookup()` operations using the RMI registry provided with JDK1.1, and using our *FT Registry*. Experiments were conducted in the same setting as in Section 4.2. The results are shown in Figure 7.

Our implementation of `lookup()` is fast, even when it provides a richer functionality. On the other hand, our `bind()` operation is

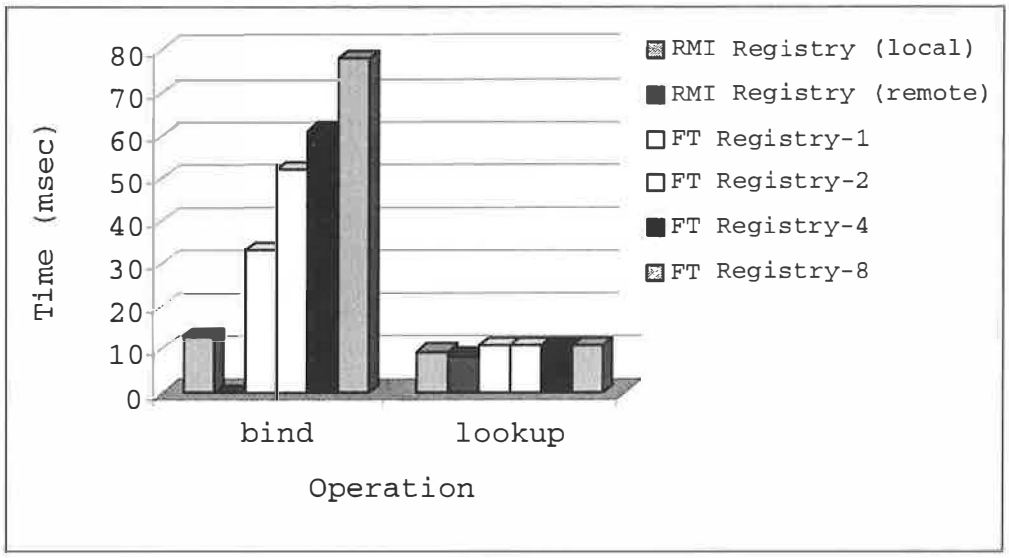


Figure 7: Performance of *FT Registry*. The x-axis contains the `lookup()` and `bind()` operations for both the standard RMI registry and our fault tolerant implementation. The y-axis represents time in milliseconds.

slower, since such events must be sent to all replicas and the new entry becomes visible at remote hosts. This functionality is not provided by the standard RMI registry service. Again, our implementation of registry service seems to scale reasonably well. For example, a bind operation with a group of 8 registry servers is approximately 2.5 times slower than a group of size 1.

6 Implementation of *FT Unicast*

On the client side of an RMI application, a stub object contains a handle for the remote object that it represents. This handle is represented by the `RemoteRef` interface. The remote reference is used to make method invocations on objects for which it is a reference. The stub object is exported by the server side to the client side. When a client makes a remote method invocation on an object through its stub, first the `newCall()` method of the corresponding remote reference is invoked by giving the remote object name and the operation in the object that needs to be performed. The `newCall()` method initiates a new connection and returns an object of type `RemoteCall` interface. Then, the `invoke()` method of the remote reference is called with this `RemoteCall` object as the parameter to execute the remote method over this connection. The remote method is executed by calling the `executeCall()` method of the `RemoteCall` object.

In order to implement *FT Unicast*, we implemented our own versions of the `RemoteRef` and the `RemoteCall` interfaces. This works as follows. When the `executeCall()` method is called in our implementation of the `RemoteCall` interface, we pass the control of execution to the underlying (and unmodified) RMI mechanism through a method call. If this call is successful, then the result from the remote method invocation is returned to the user. If the call is unsuccessful because of server failure, the existing connection that has been established inside the `RemoteRef` is released and the live reference for this failed server is acquired from the `RemoteRef`. Then, the local RMI registry is contacted first with this live reference to get the name of the server, and then again with the name of the server to get a new live reference for another replicated server. These functionalities are provided by our distributed RMI registry through the implementation

of `reverseLookup()` and `lookupLiveRef()` methods. The latter method, if given a replicated server name as a parameter, randomly returns a live reference for some replica of the server. We do not need to get a complete object reference for the server as we already have a stub for the server. We only need a live reference to establish a connection to another available replica of the server. Then the old live reference (of the now unavailable server) at the `remoteRef` is replaced by the new reference using the `setRef()` method of the corresponding `remoteRef`. Again the process is repeated by making the `RemoteRef` establish a new connection, instantiate a new `RemoteCall` object and then calling the `invoke()` method, until the remote method invocation is successful. Then the results are returned to the client.

The process explained above is executed transparently to the client. That is, the client makes only a single method invocation on a remote object server and if this server is unavailable, the remote reference layer masks this failure by finding an available server, executing the method and eventually returning the results of this method invocation to the client.

As we mentioned earlier, we have our own implementation for the `RemoteRef` interface. Because this handle is exported from the server side, we need to make sure that this handle is correctly bound to our implementation before it is exported from the server side. This is done as follows. In Java RMI, the server object inherits from the `UnicastRemoteObject` which is an extension of a remote server. Instead, we have our own extension that mirrors the `UnicastRemoteObject` which we call the `FTUnicastRemoteObject`. In our case, when the server implementation is instantiated, the `exportObject()` method of the corresponding `FTUnicastRemoteObject` is called. This method instantiates an object of class `FTUnicastServerRef` and calls the `exportObject()` method of this object. This method sets the skeleton to the proper skeleton class, the stub to the proper stub class by setting the `RemoteRef` (which in our case is of type `FTUnicastRef`) correctly, and creates a binding between the remote object and the stub. The stub object is then exported to the client side.

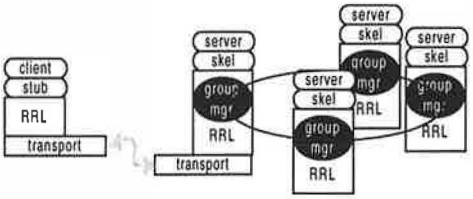


Figure 8: Highly available server architecture

In the next section, we discuss how the `GroupManager` class can be used to build fault-tolerance into any general purpose Java application server through replication.

7 Implementing Highly Available Application Servers

Our implementation of the `FTUnicastRemoteObject` class enables a client to transparently recover from server failures. It works by allowing multiple servers to register under the same name, and in the event of a failure, it redirects a client's request to another server. This method only works however, for *state-less* servers. That is, servers that do not modify their state based on client requests. An HTTP server is an example of a state-less server. State-full servers on the other hand require a general solution that is not provided by *FT Unicast*. In this section we address the general problem by integrating our `GroupManager` and `FTUnicastRemoteObject` class implementations, and provide a general architecture.

The `GroupManager` class has so far been used to construct the highly-available *FT Registry*, a state-full server. This concept can be generalized to make any application server fault-tolerant—by replicating the server and using `GroupManager` class to manage replicas. An architecture for such a highly available server is shown in Figure 8. In this case, the group managers ensure reliable ordering

of events across all the server replicas and guarantee that servers have a consistent state. Failure detection of servers can be performed, as in the *FT Registry*, by the group managers pinging each other in the background. Similarly, dynamic addition of server replicas can be allowed by transferring the state of an existing server to the newly added replica.

The ability to detect server failures, and to transparently redirect a client's request to a replicated server is another key ingredient in our design. We have already demonstrated that this functionality can be integrated within the Java RMI architecture at the RRL level, by implementing `FTUnicastRemoteObject` class. But unlike the `FTUnicastRemoteObject` class, here the client has the illusion of a single server but in reality there are replicated servers that are coordinated by the group managers.

Currently, we are implementing a `FTMulticastRemoteObject` class that can be used in place of the `UnicastRemoteObject` class provided by JDK1.1. The `FTMulticastRemoteObject` class will enable replicated servers to provide the illusion of a single server to a client. A server that inherits this class will become a member of a multicast group and any remote method calls to this server object will be multicast to all the replicas in the group.

8 Conclusions

In this paper we presented the design of *Filterfresh*, a Java package that provides support for building fault-tolerance into replicated Java server objects by implementing an underlying *Group Communication* mechanism. We described the `GroupManager` class that is instantiated with each replica and implements the group communication mechanism. We showed how the `GroupManager` class can be used to construct a fault-tolerant RMI registry server – *FT Registry*. We also described the *FT Unicast* mechanism that enables application server failures to be tolerated at the client stub layer, transparent to the client, using the *FT Registry*. Future work includes completing the implementation of the `FTMuliticastRemoteObject` class that enables the group manager support to be general so that it can be used to make any application server highly available and also extensions to *Filterfresh* to support nested invocations which will be required in this case.

References

[1] G. Beedubail, A. Karmarkar, A. Gurijala, W. Marti, and U. Pooch. An Algorithm for Supporting Fault Tolerant Objects in Distributed Object-Oriented Operating Systems. In *Proc. Fourth International Workshop on Object-Oriented Operating Systems*, 1995.

[2] K. Birman, A. Schiper and P. Stephenson. Light-weight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, August 1991.

[3] K. Birman and R. Van Renesse. *Reliable Distributed Computing with ISIS Toolkit*, IEEE Computer Society Press, 1994.

[4] N. Brown, C. Kindel. Distributed Component Object Model Protocol – DCOM/1.0. *Internet Draft*, 1996.

[5] T. Chandra, V. Hadzilacos and S. Toueg. Impossibility of group membership in asynchronous systems. Technical Report 95-1533, Computer Science Department, Cornell University, August 1995.

[6] D. Dolev, D. Malki, and R. Strong. A Framework for Partitionable Membership Service. In *Proc. of the 15th Annual ACM Symposium on Principles of Distributed Computing*, 1996.

[7] M. Fischer, N. Lynch and M. Peterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, April 1985.

[8] S. Hirano, Y. Yasu, and H. Igarashi. Performance Evaluation of Popular Distributed Object Technologies for Java. In *Proc. of ACM Workshop on Java for High-Performance Network Computing*, 1998.

[9] IONA Technologies. *http://www-usa.iona.com/Press/PR/Isis.html.*

[10] W. Jia. Implementation of a Reliable Multicast Protocol. *Software Practices and Experience*, July 1997.

[11] M. Kaashoek. *Group Communication in Distributed Computer Systems.* Ph.D Thesis, Vrije Universiteit, Netherlands, 1992.

[12] M. Kaashoek, A. Tanenbaum, and K. Verstoep. Using Group Communication to Implement a Fault-Tolerant Directory Service. In *Proc. of the 13th International Conference on Distributed Computing Systems*, 1993.

[13] S. Maffeis. Adding Group Communication and Fault-Tolerance to CORBA. In *Proceeding of USENIX Conference on Object-Oriented Technologies*, June 1995.

[14] S. Maffeis. A Fault-Tolerant CORBA Name Server. In *Proc. of Symposium on Reliable Distributed Systems*, 1996.

[15] S. Maffeis. iBus – The Java Intranet Software Bus. *http://www.softwired.ch/ibus.htm.*

[16] L. Moser, P. Melliar-Smith, D. Agarwal, R. Budhia and C. Lingley-Papadopoulos. Totem: A Fault-Tolerant Multicast Group Communication System". *Communications of the ACM*, vol. 39, no. 4, pp. 54–63, April, 1996.

[17] Object Management Group. *The Common Object Request Broker: Architecture and Specification 2.1*, 1997.

[18] F. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, December 1990.

[19] R. Stevens. *UNIX Network Programming*, Prentice Hall, 1990.

[20] Sun Microsystems. *Remote Method Invocation Specification*, 1997. *http://java.sun.com/products/jdk/1.2/docs/guide/rmi/index.html.*

[21] Y. Wang, Y. Huang, K. Vo, E. Chung and C. Kintala. Checkpoint and its applications. In *Proceedings of the 25th IEEE Fault Tolerant Computing Symposium*, June 1995.

[22] A. Wollrath, R. Riggs and J. Waldo. A Distributed Object Model for the Java System. *USENIX Journal*, Fall 1996.

COMERA: COM Extensible Remoting Architecture

Yi-Min Wang
Microsoft Research
Woei-Jyh Lee
New York University

Abstract

In a distributed object system, remoting architecture refers to the infrastructure that allows client programs to invoke methods on remote server objects in a transparent way. In this paper, we study the strength and limitations of current remoting architecture of COM (Component Object Model), and propose a new architecture called COMERA (COM Extensible Remoting Architecture) to enhance the extensibility and flexibility. We describe several application scenarios and implementations to demonstrate the power of such a componentized remoting architecture.

1. Introduction

Distributed object systems such as DCOM [Brown96], CORBA [CORBA95][Vinoski97], and Java RMI [Wollrath95], have become increasingly popular. In essence, they provide the infrastructure for supporting remote object activation and remote method invocation in a client-transparent way. A client program obtains a pointer (or a reference) to a remote object, and invokes methods through that pointer as if the object resides in the client's own address space. The infrastructure takes care of all the low-level issues such as packing the data in a standard format for heterogeneous environments (i.e., marshaling and unmarshaling), maintaining the communication endpoints for message sending and receiving, and dispatching each method invocation to the target object. In this paper, we use the term ***remoting architecture*** [COM95] to refer to the entire infrastructure that connects clients to server objects.

In general, a distributed object system does not have to specify how the remoting architecture should be structured. It can be treated as a black box as far as user applications are concerned. This black-box approach has the advantage of allowing vendors to put in their best performance optimization techniques. A disadvantage is that such architectures are usually not extensible. As a result, when low-level system properties such as load-balancing and fault tolerance are desirable, they need to be either tightly integrated with the infrastructure [Maffeis95] or provided through interception mechanisms outside the infrastructure [Narasimhan97].

In this paper, we propose an extensible remoting architecture and demonstrate that it facilitates the incorporation of low-level system properties into the infrastructure and allows them to be customized in a flexible way. We use COM's remoting architecture [COM95] [Brown96] as a starting point for the following two reasons. First, it has ***built-in extensibility***. By supporting a mechanism called ***custom marshaling***, COM allows a server object to bypass the standard remoting architecture and construct a custom one without requiring source code modifications to the former. Second, it is ***componentized***. COM's remoting architecture not only provides the basis for building distributed component-based applications, but also can be a distributed component-based application by itself. More specifically, the remoting architecture is constructed at run time by instantiating and connecting various dynamic components, and so a custom architecture can reuse some of the binary components from the standard one.

We point out the limitations of current COM remoting architecture, and propose a truly componentized, extensible architecture called COMERA. The approach is to use custom marshaling to implement COMERA, and then use COMERA to implement the low-level system properties. Three application categories are used to demonstrate the flexibility provided by COMERA: ***configurable multi-connection channels*** allow clients to use a single pointer to transparently talk to multiple server objects for performance or fault tolerance; ***transport replacement*** allows applications to run DCOM on any transport by wrapping protocol-specific client and server programs as COM objects and plugging them into COMERA; ***client-transparent object failover and migration*** allows a client to use an existing pointer to reach an object that has moved to another machine.

The paper is organized as follows. Section 2 gives the background for COM, and describes the current COM remoting architecture. Section 3 discusses the limitations of current architecture, presents the new COM-ERA architecture and specifies the interactions among its components. Section 4 describes the three application categories. Section 5 surveys related work, and Section 6 summarizes the paper.

2. Component Object Model

2.1. Overview of COM

In COM, an ***interface*** is a named collection of abstract operations (or methods) that represent one functionality. An ***object class (or class)*** is a named concrete implementation of one or more interfaces. An ***object instance (or object)*** is an instantiation of some object class. An ***object server*** is an executable (EXE) or a dynamic link library (DLL) that is responsible for creating and hosting object instances. A ***client*** is a process that invokes a method of an object. Figure 1 shows a client holding a pointer to one of the interfaces of an object. Each interface of an object represents a different view of that object and is identified by a 128-bit globally unique identifier (GUID) called the ***interface ID (IID)***. The object server contains multiple object instances from different classes, each of which is identified by a GUID called the ***class ID (CLSID)***. COM objects are usually created by class factories, which are themselves COM objects with standard interfaces for creating other COM objects.

COM specifies a binary standard that objects and their clients must follow to ensure dynamic interoperability. Specifically, any COM interface must follow a standard memory layout, which is the same as the C++ virtual function table [Rogerson96][Box98]. This allows COM applications to reuse binary code at run time through the client/server relationship, in contrast with the common notion of source code reuse at compile time. In addition, any COM interface must inherit from the IUnknown interface, which consists of a QueryInterface() call for navigating between interfaces of the same object, and two calls AddRef() and Release() for reference counting.

2.2. Remoting architecture

Figure 2 shows the current COM remoting architecture. The initial mechanism by which the client connects to the server is not shown. It can be an object activation call such as CoCreateInstance(), a binding call through a moniker (specifying a CLSID as well as particular persistent data) [Chappel96], or a lookup from a naming service. When the server object is created and is about to *export* an interface pointer, COM run-time will ask the object if it supports an IMarshal interface. If no such interface is supported, COM starts the following standard marshaling process [Chung97]:

- A ***standard marshaler*** is invoked to *marshal the interface pointer*, i.e., to pack sufficient information in an ***object reference (OBJREF)*** to be shipped to the client so that the client can use the remote pointer in a transparent way.
  - The standard marshaler loads and creates an interface stub according to the requested IID. An ***interface stub*** is itself a COM object that knows how to unmarshal input parameters and marshal output parameters for all method calls of an interface identified by a particular IID.
  - The standard marshaler gives *the pointer to the created interface stub* to a stub manager, and gets back an ***Interface Pointer ID (IPID)***. The ***stub manager*** will be in charge of dispatching each client call to the target interface stub based on the IPID tagged to the call.
  - The standard marshaler packs the IPID, the communication endpoint information (e.g., RPC string binding), and other information into a standard OBJREF and gives it to COM run-time.
- COM run-time ferries all the information obtained from the marshaler to the client side, activates a standard unmarshaler, and hands it the OBJREF.
- The standard unmarshaler creates a standard ***object proxy***, which serves as the proxy for all IUnknown method calls to the remote object. If the requested interface is not IUnknown, the object proxy also loads and creates an appropriate ***interface proxy***, aggregates it [Rogerson96], and exposes the interface of the interface proxy as if it is the object proxy's own interface.
- The object proxy also loads and creates a standard ***RPC channel object***, and uses the information in OBJREF to initialize the channel. The channel object can then use the communication endpoint information to reach the server. It also tags each call with the associated IPID so that it

can be properly dispatched once it reaches the server.

- Finally, COM run-time returns to the client an interface pointer. If it's an IUnknown pointer, it points to the object proxy; otherwise, it points to an interface proxy aggregated into the object proxy.

Once the standard remoting architecture is established, the client can make calls through the obtained pointer. Each call first enters a proxy, gets appropriately marshaled in the Network Data Representation (NDR) format [DCE95], sent by the channel object to the server endpoint, dispatched by the stub manager, gets unmarshaled by an interface stub, and finally delivered to the server object. Since the entire process is a sequence of local and remote function calls, the reply is done by simply returning from those function calls and reversing the marshaling procedure.

An object can declare that it wants to implement custom marshaling by supporting the IMarshal interface. In this case, the standard remoting architecture is not created. Instead, the object specifies (or itself acts as) a ***custom marshaler*** that is responsible for constructing ***custom OBJREFs*** and specifying the CLSID of a ***custom unmarshaler*** to be activated at the client side to receive the OBJREFs. The custom unmarshaler can create (or itself act as) a ***custom proxy*** that does application-specific processing and uses a application-specific communication mechanism to interact with the server.

3. Extensible Remoting Architecture

3.1. Extensibility issues in current COM remoting architecture

Custom marshaling provides the basis for extensibility in COM remoting architecture. Applications can achieve stronger low-level system properties by plugging in their own custom remoting architecture without having to modify the source code of the standard one. However, most such applications do not want to rebuild the entire remoting architecture; instead, they often want to reuse existing architecture as much as possible and replace only those parts that are specific to them. For example, very few applications need to replace the interface proxies and stubs that do marshaling and unmarshaling; some applications need to replace only the client-side architecture, while some need to modify only the server-side architecture. The examples described in the next section illustrate these different requirements.

The above discussion motivates the concept of a *componentized remoting architecture*. In addition to providing the infrastructure for higher-level componentized applications, if the remoting architecture itself is also componentized, then the benefits of software reuse can also be realized at the lower level. Figure 2 shows that current COM remoting architecture is partially componentized: the proxies, channels, stubs, and marshalers are COM components, but the server endpoints and stub managers are not. This limitation makes it hard to replace the transport and to control the IPID assignment and call dispatching. The second limitation is a result of the intimacy between object proxy and channel object. Although they interact through COM interfaces, this intimacy makes it hard to replace one of them without replacing the other. Specifically, the CLSID of the standard RPC channel object is not published, so it is hard for a custom proxy to connect to a standard channel. Also, the object proxy always creates and connects to a standard channel, so it is hard to reuse the object proxy while replacing the channel object.

Figure 2 illustrates the strength and the weakness of current COM remoting architecture in terms of extensibility. Basically, the architecture is extensible only at the upper layer where it interfaces to the client and the server applications. Applying custom marshaling at this layer is usually called ***semi-custom marshaling*** (or handler marshaling) as it essentially builds custom marshaling on top of standard marshaling. An arbitrary number of components connected in an arbitrary way can be inserted between the server object and the interface stubs as part of the *interface pointer marshaling process*. Such extensibility is useful for parameter tracing and logging, input value checking, etc. Similarly, arbitrary components can be inserted between the client and the proxies as part of the *interface pointer unmarshaling process*. This can be an ideal place for data caching logic, for example. In contrast, current COM remoting architecture has limited flexibility for applications that require extensibility at the lower layers. For example, fault tolerance mechanisms often need to get access to call parameters in their marshaled format for efficient logging or replication. Currently, that would require rebuilding the entire remoting architecture.

3.2. The COMERA architecture

We propose a new architecture called ***COMERA (COM Extensible Remoting Architecture)*** to address the above issues. Our approach is to first use custom marshaling to rebuild the standard remoting architecture, and then redesign parts of it to enhance extensibility. Figure 3 shows the overall COMERA architecture. Since the original interface proxies and stubs are packaged as binary COM objects, COMERA can reuse them without requiring a new IDL compiler or any recompilation. COMERA improves upon current COM remoting architecture in the following aspects:

- COMERA stub manager is a COM object and that offers two advantages. First, applications can replace the stub manager with a custom one to control IPID assignment and call dispatching. Second, COMERA marshaler interacts with the stub manager through a specified COM interface, and so replacing stub manager does not require the marshaler to be replaced as well.
- COMERA endpoint is also a COM object. It can be replaced to enable pre-dispatching message processing such as message logging and decryption. Since it provides communication endpoint information through a specified COM interface, a custom endpoint object can work with a COMERA marshaler.
- COMERA extends the IMarshal interface to include one more method call GetChannelClass() that allows a server object to specify the CLSID of a custom channel. When COMERA object proxy receives the OBJREF (standard or custom), it creates a channel object of the specified CLSID and initializes it with the OBJREF through a specified COM interface.
- The CLSID of the COMERA RPC channel object is specified so that any custom proxy can reuse this standard channel.

4. Applications

In this section, we describe three application categories to illustrate the benefits of COMERA's componentized remoting architecture. Configurable multi-connection channels enable dynamic transparent fault tolerance and support the notion of Quality-of-Fault-Tolerance. Transport replacement facilitates the low-level manipulation of marshaled data stream. Finally, the ability to restore server-side communication and dispatching state makes it possible to implement transparent object failover and migration.

4.1. Configurable multi-connection channels

A generic mechanism for transparent fault tolerance is for a client-side infrastructure to connect to multiple equivalent servers. The infrastructure can then mask server failures by retrying another server when one fails, or by sending each request to multiple servers simultaneously. This can be implemented on current remoting architecture using semi-custom marshaling as follows. The server object IMarshal routine packs multiple standard OBJREFs (corresponding to multiple equivalent objects) into one custom OBJREF; a custom proxy extracts and unmarshals each standard OBJREF into a pair of object proxy and standard channel connecting to one of the objects. Clearly, this is not efficient because the object proxy and the aggregated interface proxies are unnecessarily duplicated. Also, the marshaling and unmarshaling routines may be unnecessarily executed multiple times.

Figure 4 shows how COMERA allows the above fault tolerance mechanism to be implemented in a more natural and efficient way. In addition to packing multiple standard OBJREFs into a custom OBJREF, the server object also specifies the CLSID of a configurable multi-connection channel object. Connections to multiple objects are encapsulated inside the custom channel instead of a custom proxy. This allows the same marshaled data stream to be shared to reduce both time and memory overhead. Such architecture can also be used to provide *configurable timeouts*, which is currently not supported by COM.

We have implemented a system based on the architecture shown in Figure 4 to support ***Quality-of-Fault-Tolerance (QoFT)***. A server object dynamically determines the level of fault tolerance that should be provided for each client when the client first connects to the object, based on the client's login account. If it is a base-level client, standard remoting architecture without any fault tolerance is established. Otherwise, the determined level and the OBJREFs of the chosen server objects are transmitted to the client side to initialize a QoFT channel, of which the CLSID is also specified by the server. A level-1 client normally connects to a primary object, but will switch to a backup object when the primary server fails and the call times out. For a level-2 client, every call is sent to multiple objects and the first response is delivered to the client. This approach masks failures as well as improves response time. The system also sup-

ports *dynamic code downloading*: the DLL code of the QoFT channel object can be downloaded to the client as part of the custom marshaling stream. A custom unmarshaler is activated to extract the code and register it with the registry so that a QoFT channel object can be instantiated from it. This feature allows a service provider to try out different QoFT plans without requiring clients to install new software.

4.2. Transport replacement

According to the specification [Brown96], DCOM runs on top of RPC. In turn, RPC can run on top of different transports. For example, on Windows NT, Microsoft RPC can be configured to run on TCP, UDP, NetBIOS, and IPX by simply changing a registry setting [Nelson97]. In addition to this flexibility, applications may wish to replace the standard RPC channel altogether for a number of reasons. For example, running DCOM on HTTP may be necessary for passing through certain firewalls. Some organizations may need to run DCOM on proprietary transports in order to interoperate with existing legacy systems. Some applications may require encrypted channels for additional security. Information-dissemination applications may want to replace unicast channels with multicast channels for efficiency. Transport replacement can of course be accomplished on current COM architecture by using custom marshaling, but that would generally require rebuilding the entire remoting architecture.

Figure 5 illustrates how a new channel can be plugged into COMERA without modifying the upper layer of the architecture. The custom endpoint object wraps the transport-specific server code with a COM interface. It hosts the server-side communication endpoint and supplies binding information to the marshaler. The custom channel object wraps the transport-specific client code with two COM interfaces: one for initialization with the binding information and one for the actual communication. Some transports exist in the form of protocol stacks and can be completely wrapped inside the two COM objects. Others may require the channel object to connect to a client-side daemon, which is connected to a server-side daemon that is in turn connected to the endpoint object.

4.3. Object migration

Object migration is a generic mechanism for load balancing and fault tolerance. The goal is to move an object to another machine while still allowing existing clients to connect to it. On the one hand, the remoting architecture facilitates implementing object migration in a transparent way by providing a natural hiding place for the migration logic. On the other hand, the abstraction that it provides to the applications may hide too many low-level details, which makes transparent object migration difficult. Specifically, an object instance is uniquely identified by an RPC string binding (containing an IP address and a port number) and an IPID. Since current COM remoting architecture hides the assignment of port numbers and IPIDs from the applications, it is difficult to migrate an object while maintaining the same port number and IPID so that existing client-side channels can still reach the migrated object.

With current architecture, transparent object migration can be implemented using semi-custom marshaling as follows. A custom proxy containing the migration logic is inserted between the client and the object proxy. When a migration occurs, the custom proxy either gets notified through a special callback interface or detects that when a call times out. It then queries a migration manager process that maintains a mapping between pre-migration OBJREF and post-migration OBJREF for each migrated object. When the custom proxy gets back the new OBJREF, it creates a new pair of object proxy and channel object to connect to the migrated object, and discards the original pair.

The COMERA architecture facilitates transparent object migration in two ways. First, similar to the discussions in Section 4.1, object migration should involve only channel objects but not object proxies. By pushing the migration logic from the proxy level down to the channel level, COMERA allows a custom channel to simply update its RPC binding to connect to the migrated object, without any object activation and deactivation overhead. Second, it is particularly useful for implementing transparent object failover, which is a special case of object migration where the migrated-to machine has the same IP address as the original one. Figure 6 shows how COMERA supports transparent failover without requiring any custom objects or migration logic on the client side. The failover of the IP address can be provided by commercial clustering software [NTMag97]. The failover endpoint object checkpoints and restores the RPC string bindings. The failover stub manager checkpoints and restores IPID assignments. Since the

RPC layer has a built-in reconnection capability when an existing connection is broken, the client will be able to automatically reach the failed-over object (with the same IP, port number and IPID) upon the reconnection.

5. Related Work

CORBA does not specify a standard remoting architecture. As a result, incorporating stronger system properties such as fault tolerance into CORBA-based systems is usually not done by exploiting the extensibility in the remoting architecture. Instead, three other approaches have been taken [Narasimhan197]. *Electra* [Maffeis95] and *Orbix+Isis* [Landis97] build the mechanisms for object replication and consistency management into the ORB itself. The *Eternal* system [Narasimhan97] intercepts IIOP-related system calls through the Unix /proc interface, and maps them to routines supported by a reliable multicast group communication system. In contrast with the above two application-transparent approach, a third approach is to provide fault tolerance through a CORBA-compliant *Object Group Service* [Felber96].

COM currently supports a *channel hook* mechanism to allow piggybacking out-of-band data, which can be considered as a simple form of extensibility. By supporting an IChannelHook interface, a sender can fill in additional data to be transmitted as body extensions [Brown96] in a DCOM message, and a receiver can retrieve each body extension using its unique ID. Iona Orbix allows eight *filters* to be inserted at different places to get access to marshaled or unmarshaled call parameters or return parameters [Iona96]. Similar capabilities can be implemented on COMERA through component insertion or replacement.

The newly announced *COM+* runtime and services [Kirtland97] promise to provide a general extensibility mechanism called *interceptors*. The interceptors are used to interpret special class attributes, to receive events related to object creation/deletion and method invocation, and to automatically enable appropriate services. The *Coign* runtime system [Hunt97] provides similar instrumentation capabilities for *Inter-Component Communication Analysis (ICCA)* that serves as the basis for optimal distribution of component-based applications across a network. Compared to COM+ interceptors and Coign, COMERA does not intercept object creation calls, but the architecture for component insertion and replacement is more flexible.

Legion [Grimshaw96] is a metasystems software project that aims at providing flexible support for wide-area computing. Among its top design objectives is an extensible core consisting of replaceable components for customizing mechanisms and policies. The emphasis on transparent fault tolerance, migration, and replication is similar to COMERA. The main difference is that Legion targets high-performance parallel computing, while COMERA places more emphasis on client-server based systems.

The *Globe* project [Homburg96] proposed an architecture for distributed shared objects. Each local object consists of four subobjects: a control object handling local concurrency control; a semantics object providing the actual semantics of the shared object; a replication object responsible for state consistency management; and a communication object that handles low-level communication. COMERA can be used to support this architecture by implementing the control and the semantics objects in a custom proxy, and the replication and the communication objects in a custom channel.

6. Summary

We have proposed COMERA as an extensible remoting architecture for COM. By componentizing the architecture into COM objects, COMERA makes the low-level distributed objects infrastructure itself as dynamic, flexible, and reusable as the applications that it supports. We used three application categories as examples to demonstrate the advantages of the new architecture. For the multi-connection channels category, we have implemented a Quality-of-Fault-Tolerance subsystem on top of COMERA to support dynamic determination of fault-tolerance levels and dynamic code downloading for custom proxies and channels. For the transport replacement category, we have successfully plugged a commercial reliable multicast protocol implementation into COMERA. A programming wizard can be provided to further simplify the tasks of channel and endpoint object wrapping. For the object migration category, we have implemented a transparent failover subsystem for COM objects on top of IP failover. Future work includes building active replication and distriuted shared objects on COMERA.

Acknowledgement

The authors would like to express thanks to Li Li (Cornell) for his contributions to the initial imple-

mentation of COMERA, to Emerald Chung, Chung-Yih Wang, and Yennun Huang from Lucent Technologies for their valuable discussions, and to the software architects and engineers from the DCOM mailing list for their positive feedback on the architecture.

References

[Box98] D. Box, Essential COM, Addison-Wesley, 1998.

[Brown96] N. Brown, C. Kindel, Distributed Component Object Model Protocol -- DCOM/1.0, http://www.microsoft.com/oledev/olecom/draft-brown-dcom-v1-spec-01.txt.

[Chappell96] D. Chappell, Understanding ActiveX and OLE, Redmond, Washington: Microsoft Press, 1996.

[Chung97] P. E. Chung, Y. Huang, S. Yajnik, D. Liang, J. C. Shih, C. Y. Wang, and Y. M. Wang, "DCOM and CORBA Side by Side, Step by Step, and Layer by Layer," in *C++ Report*, Vol. 10, No. 1, pp. 18-29,40, Jan. 1998.

[COM95] The Component Object Model Specification, http://www.microsoft.com/oledev/olecom/title.htm.

[CORBA95] The Common Object Request Broker: Architecture and Specification, Revision 2.0, July 1995, http://www.omg.org/corba/corbiiop.htm.

[DCE95] DCE 1.1: Remote Procedure Call Specification, The Open Group, http://www.rdg.opengroup.org/public/pubs/catalog/c706.htm.

[Felber96] P. Felber, B. Garbinato, and R. Guerraoui, "Designing a CORBA group communication service," in *Proc. the 15th Symp. on Reliable Distributed Systems*, pp. 150-159, Oct. 1996.

[Grimshaw96] A. Grimshaw and W. A. Wulf, "Legion," in *Proc. the Seventh ACM SIGOPS European Workshop*, Sep. 1996.

[Homburg96] P. Homburg, M. van Steen, and A. S. Tanenbaum, "An architecture for a wide area distributed system," in *Proc. the Seventh ACM SIGOPS European Workshop*, Sep. 1996.

[Hunt97] G. C. Hunt and M. L. Scott, "Coign: Efficient instrumentation for inter-component communication analysis," Tech Report 648, Dept. of Computer Science, University of Rochester, Feb. 1997.

[Iona96] Orbix 2.1 Programming guide, Iona technologies Ltd., http://www.iona.com/.

[Kirtland97] M. Kirtland, "Object-oriented software development made simple with COM+ runtime services," *Microsoft Systems Journal*, Vol. 12, No. 11, pp. 49-59, Nov. 1997.

[Landis97] S. Landis and S. Maffeis, "Building reliable distributed systems with CORBA," *Theory and Practice of Object Systems*, John Wiley & Sons, New York, 1997.

[Maffeis95] S. Maffeis, "Adding group communication and fault-tolerance to CORBA," in *Proc. Usenix Conf. on Object-Oriented Technologies*, June 1995.

[Narasimhan197] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith, "The interception approach to reliable distributed CORBA objects," in *Proc. the 3rd Conf. on Object-Oriented Technologies and Systems*, June 1997.

[Narasimhan97] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith, "Exploiting the Internet Inter-ORB Protocol interface to provide CORBA with fault tolerance," in *Proc. the 3rd Conf. on Object-Oriented Technologies and Systems*, June 1997.

[Nelson97] Michael Nelson, "Using Distributed COM with Firewalls", http://www.wam.umd.edu/~mikenel/dcom/dcomfw.htm, 1997.

[Rogerson96] D. Rogerson, Inside COM, Redmond, Washington: Microsoft Press, 1996.

[Vinoski97] S. Vinoski, "CORBA: Integrating diverse applications within distributed heterogeneous environments," in *IEEE Communications*, vol. 14, no. 2, Feb. 1997. http://www.iona.com/hyplan/vinoski/ieee.ps.Z.

[NTMag97] "Clustering Solutions for Windows NT," *Windows NT Magazine*, pp. 54-95, June 1997.

[Wollrath95] A. Wollrath, R. Riggs and J. Waldo, "A Distributed Object Model for the Java System," *USENIX Journal, Computing Systems*, Vol. 9, No. 4, pp.265-289, Fall 1996.

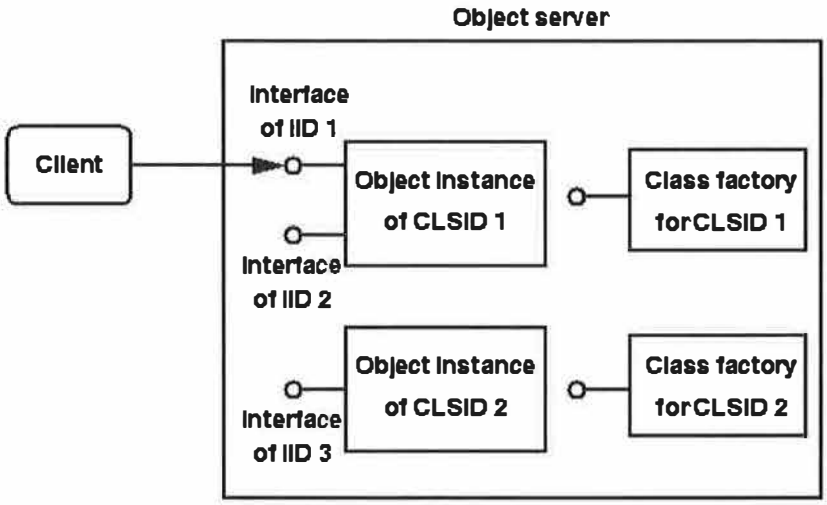


Figure 1. COM server, classes, objects, interfaces and client.

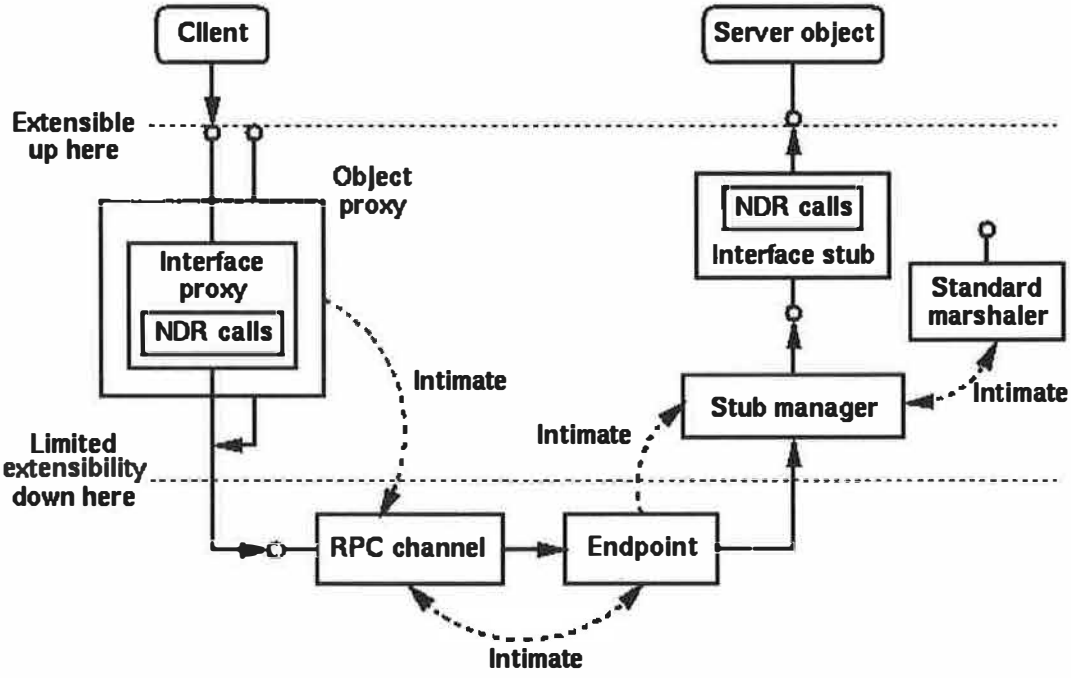


Figure 2. Limitations of current COM remoting architecture.

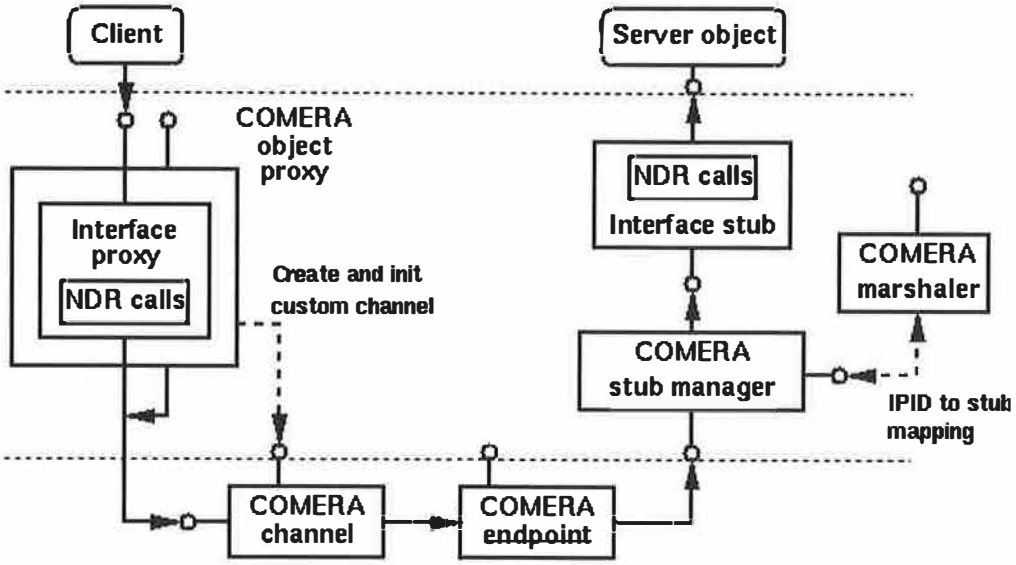


Figure 3. The COMERA architecture.

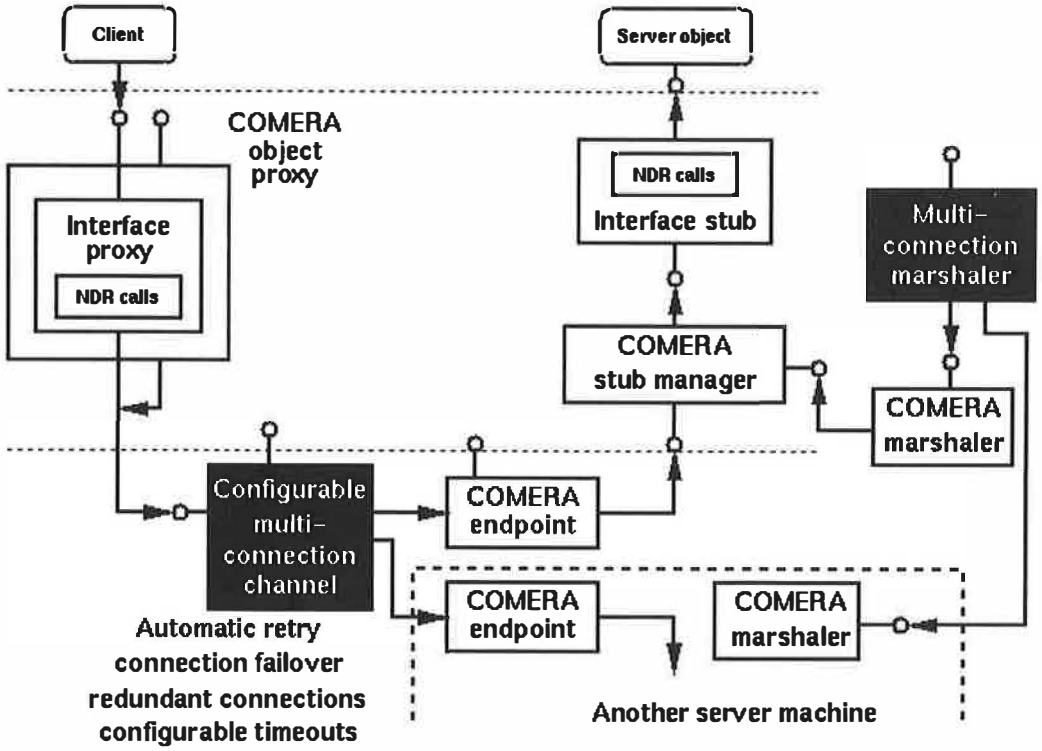


Figure 4. Configurable multi-connection channel on COMERA. (Inserted or replaced components are indicated by the black boxes.)

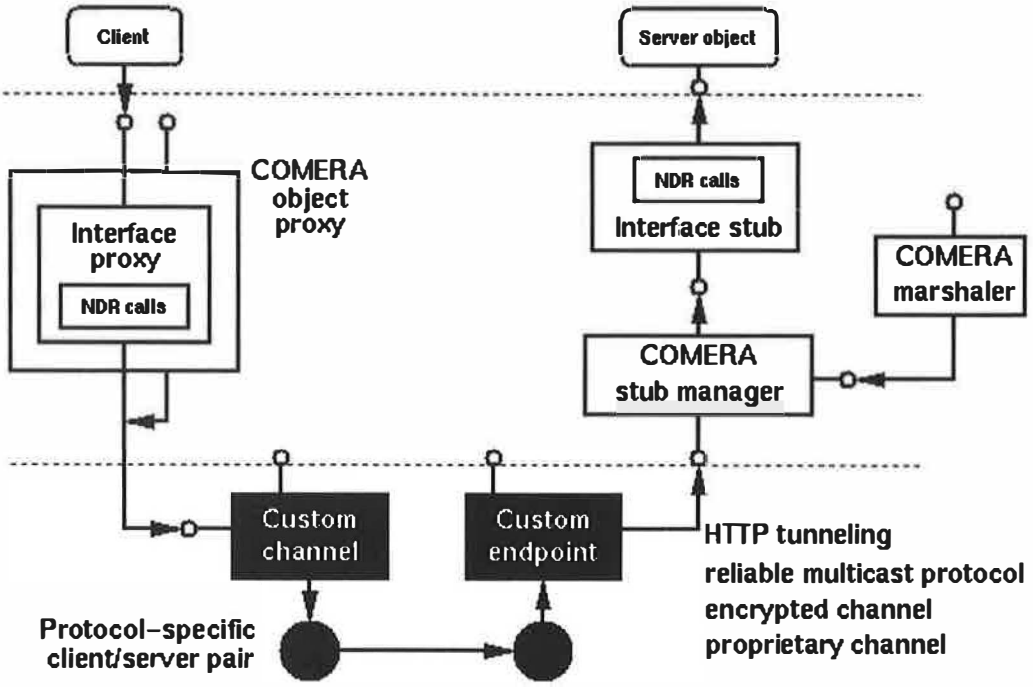


Figure 5. Transport replacement by replacing the channel and the endpoint objects.

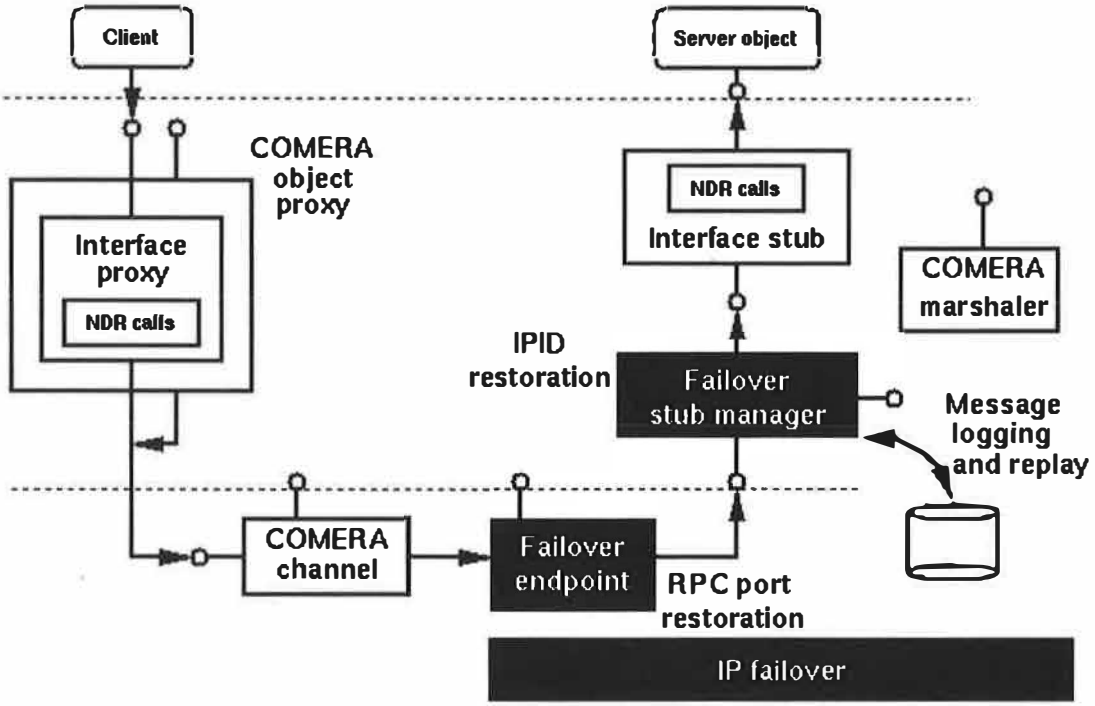


Figure 6. Client-transparent object failover with COMERA.

Java Transactions for the Internet

M.C. Little and S.K. Shrivastava
(M.C.Little@ncl.ac.uk, Santosh.Shrivastava@ncl.ac.uk)
Department of Computing Science
University of Newcastle, Newcastle upon Tyne, NE1 7RU, England

Abstract

The Web frequently suffers from failures which affect the performance and consistency of applications run over it. An important fault-tolerance technique is the use of *atomic transactions* for controlling operations on services. While it has been possible to make server-side Web applications transactional, browsers typically did not possess such facilities. However, with the advent of Java it is now possible to consider empowering browsers so that they can fully participate within transactional applications. In this paper we present the design and implementation of a standards compliant transactional toolkit for the Web. The toolkit allows transactional applications to span Web browsers and servers and supports application specific customisation, so that an application can be made transactional without compromising the security policies operational at browsers and servers.

1. Introduction

The Web frequently suffers from failures which can affect both the performance and consistency of applications running over it. For example, if a user purchases a cookie (a token) granting access to a newspaper site, it is important that the cookie is delivered and stored if the user's account is debited; a failure could prevent either from occurring, and leave the system in an inconsistent state. For resources such as documents, failures may simply be annoying to users; for commercial services, they can result in loss of revenue and credibility.

Atomic transactions are a well-known technique for guaranteeing application consistency in the presence of failures. Web applications already exist which offer transactional guarantees to users. However, currently these guarantees only extend to resources used at Web servers, or between servers; clients (browsers) are not included, despite their role being significant in applications such as mentioned previously. Providing end-to-end transactional integrity between the browser and the application is important: in the previous example, the cookie *must* be delivered once the user's account has been debited. Cgi-scripts cannot provide this level of transactional integrity since replies sent *after* the transactions have completed may be lost, and replies sent *during* the transaction may need to be revoked if the transaction cannot complete. This is an inherent problem with the original "thin" client model of the Web, where browsers were functionally barren. With the advent of Java it is now possible to consider empowering browsers so that they can fully participate within transactional applications. However, to be widely applicable, we claim that any such transaction system must meet the following three requirements:

(i) it must support distributed, nested transactions;

(ii) it must not compromise the security policy imposed at the browser's site; and,

(iii) it must comply with appropriate standards.

We have designed and implemented the *JTSArjuna* system, a transaction toolkit that meets the above requirements. Our toolkit allows transactional applications to span Web browsers and servers and supports application specific customisation, so that an application can be made transactional without compromising the security policies operational at browsers and servers. The toolkit complies with the OMG Object Transaction Service (OTS) and the Java Transaction Service (JTS) standards [OMG95][VM96]. Although the OMG has specified several object services, there is no specification for an overall object model with which to glue them together into a coherent application development framework. Therefore, we have provided a high-level API which allows programmers to be isolated from many of the issues involved in building transactional applications. This API is the result of extensive experience with the original C++ Arjuna distributed transaction system [GDP95][SKS95].

2. Transaction standards for distributed objects

For a transaction system to be widely applicable, it must conform to the standards. The most widely accepted standard for distributed objects is the Common Object Request Broker Architecture (CORBA) from the Object Management Group (OMG). It consists of the Object

Request Broker (ORB) that enables distributed objects to interact with each other, and a number of services have also been specified, which include persistence, concurrency control and the Object Transaction Service.

2.1 The Object Transaction Service

The Object Transaction Service supports the well known concept of ACID transactions. The OTS provides interfaces that allow multiple distributed objects to cooperate in a transaction such that all objects commit or abort their changes together. However, the OTS does not require all objects to have transactional behaviour. Instead objects can choose not to support transactional operations at all, or to support it for some requests but not others.

The transaction service specification distinguishes between *recoverable objects* and *transactional objects*. Recoverable objects are those that contain the actual state that may be changed by a transaction and must therefore be informed when the transaction commits or aborts to ensure the consistency of the state changes. This is achieved be registering appropriate objects that support the `Resource` interface (or the derived `SubtransactionsAwareResource` interface) with the current transaction. In contrast, a simple transactional object need not necessarily be a recoverable object if its state is actually implemented using other recoverable objects. The major difference is that a simple transactional object need not take part in the commit protocol used to determine the outcome of the transaction since it does not maintain any state itself, having delegated that responsibility to other recoverable objects which will take part in the commit process.

It is important to realise that the OTS is simply a *protocol engine* that guarantees that transactional behaviour is obeyed but does not directly support all of the transaction properties. As such it requires other co-operating services that implement the required functionality, including:

- *Persistence/Recovery Service.* Required to support the atomicity and durability properties. (There is no recovery service currently specified by the OMG.)
- *Concurrency Control Service.* Required to support the isolation properties.

2.2 Writing OTS applications

To participate within an OTS transaction, a programmer must be concerned with:

- creating `Resource` and `SubtransactionAwareResource` objects for each object which will participate within the transaction/subtransaction. These resources are responsible for the persistence, concurrency control, and recovery for the object. The OTS will invoke these objects during the prepare/commit/abort phase of the (sub)transaction, and the `Resources` must then perform all appropriate work.
- registering `Resource` and `SubtransactionAwareResource` objects at the correct time within the transaction, and ensuring that the object is only registered once within a given transaction. As part of registration a `Resource` will receive a reference to a `RecoveryCoordinator` which must be made persistent so that recovery can occur in the event of a failure.
- ensuring that, in the case of nested transactions, any propagation of resources such as locks to parent transactions are correctly performed. Propagation of `SubtransactionAwareResource` objects to parents must also be managed.
- in the event of failures, the programmer or system administrator is responsible for driving the crash recovery for each `Resource` which was participating within the transaction.

The OTS does not provide any `Resource` implementations. These must be provided by the application programmer or the OTS implementer. The interfaces defined within the OTS specification are too low-level for most application programmers. Therefore, we have designed JTSArjuna to make use of raw Common Object Services interfaces but provide a higher-level API for building transactional applications and frameworks. This API automates much of the above activities concerned with participating in an OTS transaction.

The architecture of the system is shown in figure 1. As we shall show, the API interacts with the concurrency control and persistence services, and automatically registers appropriate resources for transactional objects. These resources may also use the persistence and concurrency services.

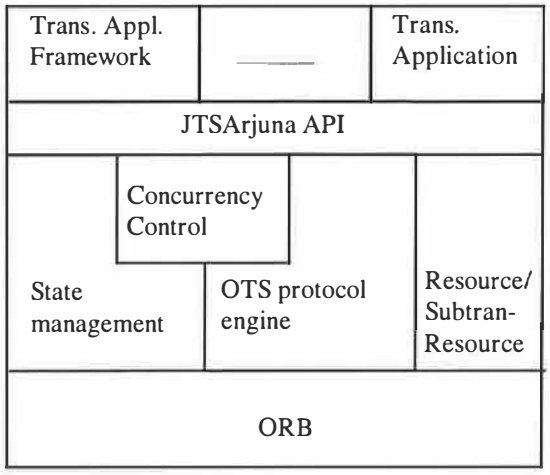


Figure 1: JTSArjuna structure

3. Requirements for configuration

The use of Java to implement transactional applications raises some important security issues. Java security is imposed by a SecurityManager object, which defines what a program can, and cannot do [DF97][JSF95]. However, there is no standard for the SecurityManager implementation, with the result that an application written for one interpreter may not be able to execute as intended on another. The recent addition of digital signatures, allowing users to specify security capabilities on a per signature basis, increases the difficulties of building truly portable applications.

The constraints imposed by SecurityManagers can directly affect transactional applications which may require, for example, to make state updates persistent by accessing the local disk. There are two obvious solutions to this problem: (i) all objects must reside within domains which have well-behaved security constraints (Web servers), or (ii) modify the Java language and the interpreter and provide an implementation of the SecurityManager which relaxes these security restrictions [MA96]. The first solution is unnecessarily restrictive in environments where SecurityManagers do allow programs increased flexibility. The second solution lacks portability as it requires users to have access to specialised implementations.

Our solution was to design and implement the *JavaGandiva* configuration support framework based on the model described in [SMW96], which isolates applications and programmers from the differences between Java SecurityManagers. Applications can be dynamically configured to take advantage of the environment in which they execute. As we shall show, several JTSArjuna classes must use this framework to provide portability across SecurityManagers.

3.1 Configuration model

Software components are split into two separate entities: the *interface component* and the *implementation component*. The interactions between implementations can only occur through interfaces. A single interface can be used to access multiple implementations, and a single implementation can be accessed through multiple interfaces. The necessity of providing multiple interfaces to implementations has long been recognised. However, we take this further by allowing the bindings of interfaces to implementations, and the interfaces an implementation can be accessed through, to be dynamic and configurable. Applications are written only in terms of interfaces, and although an application can request a specific implementation, it occurs in a way that allows this request to be changed without modifying the application. Therefore, this allows the application to be adapted for each SecurityManager by ensuring that interfaces use only those implementations which can operate within a particular environment.

3.2 JavaGandiva implementation

In an object-oriented language like Java, it is possible to map interface components and implementation components onto *interface and implementation classes* respectively. Object-orientation allows us to specify the binding between interface class and implementation class either through *inheritance* or *delegation*. We require the binding between interface classes and implementation classes to be evaluated when the interface class is instantiated. Therefore, delegation best matches our requirements to control this binding at run-time [SMW96].

In order to leave this binding until run-time we must specify it as data and not within the code of the interface class. The instance of the interface class (*interface object*) uses this data to create and bind to the correct instance of the implementation class (*implementation object*). To provide this separation of interface component and implementation component requires changing what would have been a single Java class into three classes, and a Java `interface`:

(i) *the interface class*: users interact with instances of this class, which defines the public operations that can be invoked on the implementation. The only implementation specific information present in the class definition is a reference to an instance of an *implementation interface*, to which the interface delegates all operations.

(ii) *the implementation interface*: this is a Java `interface` and all implementations accessible to an interface class implement it. This guarantees that all implementations conform to a known type.

(iii) *the implementation class*: instances of this class represent the implementation of an object. Implementation classes can be derived from multiple implementation interfaces.

(iv) *the control class*: this class provides access to operations that manipulate the non-functional characteristics of an implementation class. Implementation classes provide an operation that returns a specific instance of this control class. Interface classes provide an operation that can be used to request an instance of the implementation's control class.

Figure 2 shows an object structure formed by the above classes, where the implementation specific objects are shown in grey.

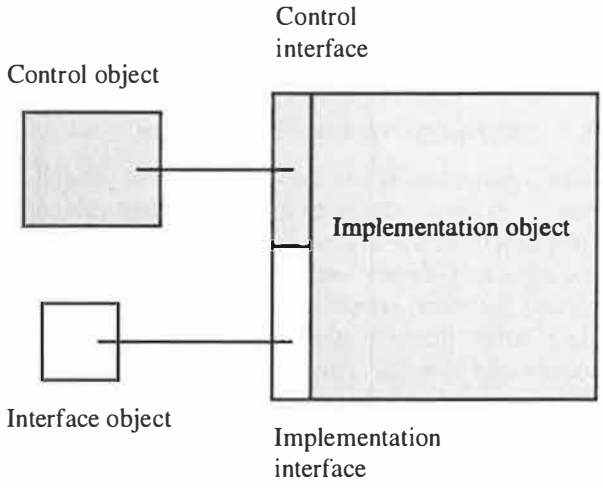


Figure 2: Interface, Implementation and Control Objects.

3.3 JavaGandiva built-time support

The JavaGandiva build-time system offers support to programmers to construct applications from existing interfaces and to build new interfaces and implementations. Interfaces can be automatically generated from a high-level definition language, and contain the necessary code to interact with the run-time system to bind to an appropriate implementation (as described in the next section).

To incorporate configurability into an application, the programmer creates a *Configuration Management Object* (CMO). The CMO contains *data* which specifies the interface to implementation bindings for the application, and any data required by implementations for initialisation. The data may also specify alternate implementations, e.g., because of possible security restrictions. At bind time an interface interrogates the CMO to determine which implementation it requires, and then passes this information to the run-time system. Importantly for our purposes, the CMO data associated with an application can be specified at run time, therefore providing a way to configure the application for each user and environment.

3.4 JavaGandiva run-time support

The run-time consists primarily of an *Implementation Repository* which is used for creating new instances of (arbitrary) implementation classes given their class names. Implementation classes can be registered with the repository so that instances of them can be created later. The repository isolates interfaces from direct implementation creation; as we shall see, all aspects of implementation creation are hidden within the repository, so that modification of the types of implementations available to an application and interface does not require changes to either.

Figure 3 illustrates how an interface uses these objects when binding to an appropriate implementation. When an interface requires to be bound to an implementation, it interrogates the application CMO for the implementation type. It then requests an instance of this type from the repository. If the requested implementation type does not exist, or cannot be used within the current environment, then the binding will fail. The interface can then attempt an alternate binding if one is specified by the CMO. Importantly, none of this is visible to the application, which simply attempts to create and use an object.

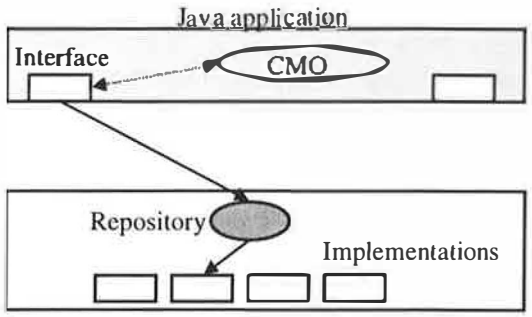


Figure 3: Application execution environment.

3.5 Specifying an application's configuration

The configuration management object is implemented by the `ObjectName` class. This configuration information is maintained as a set of *attributes*; each attribute is a name (string), value pair. An interface object uses the attributes of `ObjectName` to determine the type of its implementation; this implementation can also use the `ObjectName` to configure itself, e.g., to obtain its initial state. If multiple bindings are possible for the interface because of possible security restrictions, the `ObjectName` can specify alternate implementations.

The (simplified) signature of `ObjectName`, without the exceptions it can throw, is shown below:

```
public class ObjectName implements
                                    Serializable
{
// the supported attribute types

public static final int SIGNED_NUMBER = 0;

// for C++ compatibility
public static final int UNSIGNED_NUMBER = 1;

public static final int STRING = 2;
public static final int OBJECTNAME = 3;
public static final int CLASSNAME = 4;
public static final int UID = 5;

public int attributeType (String attrName);
public String firstAttributeName ();
public String nextAttributeName (String curr);

/*
 * Now a series of set/get methods for each
 * type of attribute. We show only two for
 * simplicity.
 */

public long getLongAttribute (String atr);
public String getStringAttribute (String atr);

public void setLongAttribute (String atr,
                              long value);
public void setStringAttribute (String atr,
                                String value);

public boolean removeAttribute (String atr);
public boolean equals (ObjectName objectName);
public boolean notEquals (ObjectName objName);

// how to store/retrieve data
private NameService _nameService;
}
```

An attribute value can be one of six basic types. `ObjectName` is responsible for run-time type checking: an exception is raised if an interface requests the wrong type for an attribute. There are methods for creating new attribute name, value pair mappings, and for retrieving an attribute given its name. Additionally, it is possible to query the type of an attribute using `attributeType`, and to iterate through all of the attributes using `firstAttributeName` and `nextAttributeName`.

To enable the configuration information to be stored in a flexible manner, `ObjectName` stores and retrieves the information using a separate NameService interface and implementation. Therefore, the means of storing this configuration data can be changed simply be changing the NameService implementation. For example, the JDBC (Java Database Connectivity) API is a standard SQL database access interface, providing uniform access to a wide range of relational databases. By providing a suitable NameService implementation, the `ObjectName` data could be maintained within such a database. However, to minimise external dependencies, our current implementation for Web applications embeds the `ObjectName` data within the HTML document which is downloaded with the Java application. The HTML document is created automatically from a separate description language.

3.6 Implementation repository

The implementation repository is provided by the Inventory, which is an interface class and a set of implementation classes. To be able to create implementations for interfaces, the inventory must be populated with these implementations. Populating the inventory can occur:

1) *statically at build time*: each implementation can be registered with the inventory when the application is built, i.e., a specific inventory is constructed for each application. If implementations are required to be added or removed from the inventory then the inventory implementation must be modified.

2) *dynamically at run time*: implementations may be loaded across the network or from the local disk. Given the name of a class, an inventory can attempt to load it dynamically. This has the advantage of flexibility, but requires the sources of these implementations (e.g., Web servers) to remain available while the application is being configured.

Because the inventory is accessed through a well-defined interface, changing the implementation from, say 1) to 2), does not require any changes in an application.

The Inventory interface class has methods for obtaining an instance of an implementation from its class name. For simplicity we show only a representative set of these methods, without the exceptions they throw:

```
public class Inventory
{
public synchronized Object createVoid
                            (String typeName);
public synchronized Object createObjectName
                            (String typeName,
                  ObjectName paramObjectName);
public synchronized Object createResources
                            (String typeName,
                     Object[] paramResources);

/*
 * A handle on the application's inventory
 * for bootstrapping (already bound interface
 * and implementation.
 */

public static Inventory inventory ();
}
```

Each create method takes the name of the implementation class to instantiate and, depending on the method, may pass additional parameter(s) to the created implementation. For example, `createObjectName` will pass the `ObjectName` parameter to the implementation when it is created. In order that the inventory can deal with any Java implementation class, it returns all created objects to the interface as instances of the Java `Object` class, which is the base class from which all Java classes are derived. The interface can then safely convert this back to the actual type.

3.7 Determining security restrictions

In order to configure itself to operate within a specific security environment, an application must be able to determine the restrictions imposed by that environment. At bind time an interface must be able to determine whether the implementation it receives from the inventory can work within the current security restrictions. Therefore, each implementation object must provide a `canExecute` method which returns either true if it can execute within the current environment, or false if it cannot. When the inventory returns an implementation object, the interface calls this method to determine whether the object can function. If it cannot, the interface can ask the `ObjectName` for the name of another implementation, and pass this to the inventory.

To determine whether or not it can function within the security environment, the implementation object may extract information from the `ObjectName` it is given when it is created, e.g., the location of the object store database to use. Shown below is the `canExecute` method for a simple object store service which writes to the local file system:

```
public SimpleObjectStore implements
                                ObjectStoreImple
{
public boolean canExecute ()
{
  /*
   * First get handle on current
   * SecurityManager.
   */

   SecurityManager manager =
                System.getSecurityManager();

   if (manager == null)
     return true; // no restrictions!
   else
   {
     /*
      * There is a SecurityManager, so
      * interrogate it.
      */

     try
     {
       /*
        * Assume these file names were read
        * from the ObjectName when we were
        * created.
        */

        manager.checkRead("/ObjStore/data");
        manager.checkWrite("/ObjStore/data);
        manager.checkDelete("/ObjStore/data");

        return true;
     }
     catch (Exception e)
     {
       /*
        * SecurityManager raised an
        * exception, could try alternate
        * location.
        */

       return false;
     }
   }
}
}
```

4. JTSArjuna implementation

JTSArjuna exploits object-oriented techniques to present programmers with a toolkit of Java classes from which application classes can inherit to obtain desired properties, such as persistence and concurrency control [MCL97]. These classes form a hierarchy, part of which is shown below.

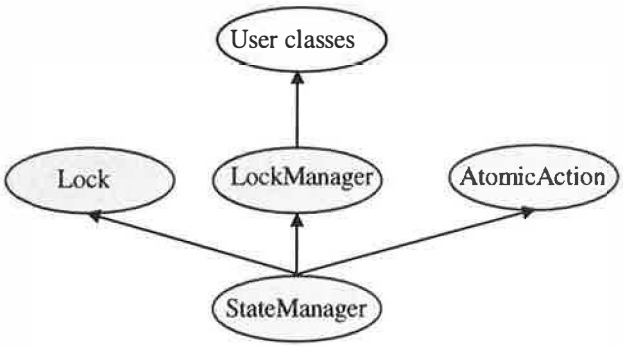


Figure 4: JTSArjuna class hierarchy

As we shall show, apart from specifying the scopes of transactions, and setting appropriate locks within objects, the application programmer does not have any other responsibilities: JTSArjuna guarantees that transactional objects will be registered with, and be driven by, the appropriate transactions, and crash recovery mechanisms are invoked automatically in the event of failures.

4.1 Saving object states

JTSArjuna needs to be able to remember the state of an object for several purposes, including recovery (the state represents some past state of the object) and persistence (the state represents the final state of an object at application termination). Since these requirements have common functionality they are all implemented using the same mechanism: the classes `InputObjectState` and `OutputObjectState`. The classes maintain an internal array into which instances of the standard types can be contiguously packed (unpacked) using appropriate `pack (unpack)` operations. This buffer is automatically resized as required should it have insufficient space. The instances are all stored in the buffer in a standard form (so-called network byte order) to make them machine independent. Any other architecture independent format (such as XDR or ASN.1) could be implemented simply by replacing the operations with ones appropriate to the encoding required. (We are currently examining using the new object serialization mechanisms within the Java language.)

4.2 The object store

Implementations of persistence can be affected by restrictions imposed by the Java SecurityManager. Therefore, the object store provided with *JTSArjuna* is implemented using the techniques of interface/implementation separation described earlier. The current distribution has implementations which write object states to the local file system or database, and remote implementations, where the interface uses a client stub (proxy) to remote services.

Persistent objects are assigned unique identifiers (instances of the Uid class), when the are created, and this is used to identify them within the object store. States are read using the `read_committed` operation and written by the `write_(un)committed` operations.

```
public interface ObjectStoreImple
{
public boolean commit_state (Uid id);
public InputObjectState read_committed (Uid
id);
public InputObjectState read_uncommitted (Uid
id);
public boolean remove_committed (Uid id);
public boolean remove_uncommitted (Uid id);
public boolean write_committed (Uid id,
                    OutputObjectState state);
public boolean write_uncommitted (Uid id,
                    OutputObjectState state);
};
```

4.3 Recovery and persistence

At the root of the class hierarchy is the class `StateManager`. This class is responsible for object activation and deactivation and object recovery. The simplified signature of the class is:

```
public abstract class StateManager
{
public boolean activate ();
public boolean deactivate (boolean commit);

public Uid get_uid (); // object's identifier.

// methods to be provided by a derived class

public abstract boolean restore_state
                        (InputObjectState os);
public abstract boolean save_state
                       (OutputObjectState os);

protected StateManager ();
protected StateManager (Uid id);
};
```

Objects are assumed to be of three possible flavours. They may simply be *recoverable*, in which case `StateManager` will attempt to generate and maintain appropriate recovery information for the object. Such objects have lifetimes that do not exceed the application program that creates them. Objects may be *recoverable and persistent*, in which case the lifetime of the object is assumed to be greater than that of the creating or accessing application, so that in addition to maintaining recovery information `StateManager` will attempt to automatically load (unload) any existing persistent state for the object by calling the `activate (deactivate)` operation at appropriate times. Finally, objects may possess none of these capabilities, in which case no recovery information is ever kept nor is object activation/deactivation ever automatically attempted.

If an object is recoverable (or persistent) then `StateManager` will invoke the operations `save_state` (while performing `deactivate`), and `restore_state` (while performing `activate`) at various points during the execution of the application. These operations *must* be implemented by the programmer since `StateManager` cannot detect user level state changes. (We are examining the automatic generation of default save_state and restore_state operations, allowing the programmer to override this when application specific knowledge can be used to improve efficiency.) This gives the programmer the ability to decide which parts of an object's state should be made persistent. For example, for a spreadsheet it may not be necessary to save all entries if some values can simply be recomputed. The `save_state` implementation for a class `Example` that has integer member variables called `A`, `B` and `C` could simply be:

```
public boolean save_state(OutputObjectState o)
{
    return (o.packInt(A) && o.packInt(B)
            && o.packInt(C));
}
```

4.4 The concurrency controller

The concurrency controller is implemented by the class `LockManager` which provides sensible default behaviour while allowing the programmer to override it if deemed necessary by the particular semantics of the class being programmed. As with `StateManager` and persistence, concurrency control implementations are accessed through interfaces. As well as providing access to remote services, the current implementations of concurrency control available to interfaces include:

- local disk/database implementation, where locks are made persistent by being written to the local file system or database.
- a purely local implementation, where locks are maintained within the memory of the virtual machine which created them; this implementation has better performance than when writing locks to the local disk, but objects cannot be shared between virtual machines. Importantly, it is a basic Java object with no requirements which can be affected by the SecurityManager.

The primary programmer interface to the concurrency controller is via the `setlock` operation. By default, the runtime system enforces strict two-phase locking following a multiple reader, single writer policy on a per object basis. Lock acquisition is (of necessity) under programmer control, since just as `StateManager` cannot determine if an operation modifies an object, `LockManager` cannot determine if an operation requires a read or write lock. Lock release, however, is under control of the system and requires no further intervention by the programmer. This ensures that the two-phase property can be correctly maintained.

```
public abstract class LockManager
                            extends StateManager
{
public LockResult setlock (Lock toSet,
                           int retry,
                           int timeout);
};
```

The `LockManager` class is primarily responsible for managing requests to set a lock on an object or to release a lock as appropriate. However, since it is derived from `StateManager`, it can also control when some of the inherited facilities are invoked. For example, `LockManager` assumes that the setting of a write lock implies that the invoking operation must be about to modify the object. This may in turn cause recovery information to be saved if the object is recoverable. In a similar fashion, successful lock acquisition causes `activate` to be invoked.

The code below shows how we may try to obtain a write lock on an object:

```
public class Example extends LockManager
{
public boolean foobar ()
{
  AtomicAction A = new AtomicAction;
  boolean result = false;

  A.begin();

  if (setlock(new Lock(LockMode.WRITE) ==
                                    Lock.GRANTED)
  {
    /*
     * Do some work, and JTSArjuna will
     * guarantee ACID properties.
     */

    // automatically aborts if fails

    if (A.commit() == AtomicAction.COMMITTED)
    {
      result = true;
    }
  }
  else
    A.rollback();

  return result;
}
}
```

4.5 Configuration hierarchy

Figure 5 shows a transactional user class inheriting from LockManager. Internally, LockManager accesses the concurrency service (CC) through an interface, and StateManager does likewise with the persistence service (POS). For each application object, the implementations of CC and POS are not chosen until run-time.

Additional implementations can be provided without changing the JTSArjuna system or applications which use it. The JTSArjuna API isolates programmers from the different POS and CC implementations, allowing them to concentrate on the application.

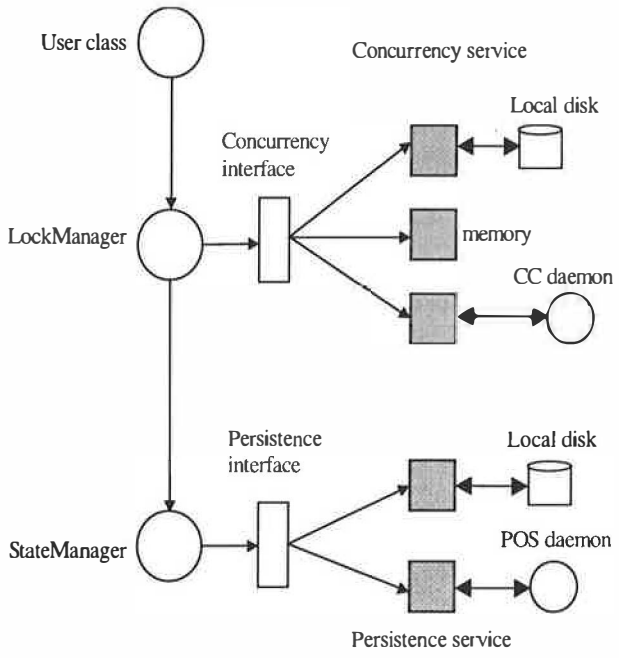


Figure 5: Configuration hierarchy

5. Performance results

Table 1 shows some basic performance results for JTSArjuna, obtained using JDK1.2 running on a Sun Ultra Enterprise 1/170 with 128Meg of RAM. In these tests, the transactional object operated upon had a single integer as its state. (As shown in the table, this transactional object was sometimes only recoverable, i.e., its state was not obtained from/saved to disk.) All timings have been averaged over 1000 runs.

Type of operation	Time taken
Update a persistent object	21.6 milliseconds
Update a recoverable object	11.2 milliseconds
Create and commit a null transaction	1.1 milliseconds
Create and commit a null nested transaction and its parent	1.9 milliseconds

Table 1:JTSArjuna performance figures

These figures represent the initial implementation of our Java transactions. Based upon our experiences with JTSArjuna and its C++ counterpart, we believe that further optimisations to the system are possible which will improve performance.

6. Newspaper example using JTSArjuna

In this section we shall illustrate the different aspects of constructing a transactional application using JTSArjuna. Consider the example of subscribing to an on-line newspaper described in the introduction.

The entities involved in the newspaper application are:

- the user's on-line bank, from where funds will be debited. We shall assume that the newspaper's account is also located here.
- the newspaper site, where the user's details will be added upon successfully completing the transaction.
- the user's browser site, where a cookie authenticating the user must be delivered and stored.

Each of the entities is represented as a separate transactional object (see figure 6). A transaction will begin when the user downloads the Java application and types in the bank account details. The application will then attempt to debit the account and, if successful, place the cookie within the cookie object at the browser. It will then commit the transaction. If a failure occurs, the transaction and all of its work will be aborted.

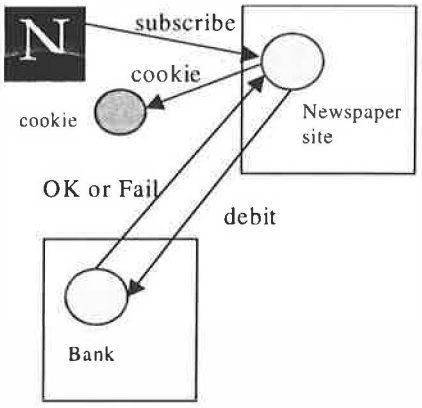


Figure 6: Transactional newspaper.

We first use the transactional toolkit to construct the application classes and partition the application as shown. An example of the cookie class which resides within the browser is:

```
public class Cookie extends LockManager
{
public Cookie ();

public boolean depositCookie (UserDetails obj)
```

```
{
  AtomicAction B = new AtomicAction();
  boolean result = false;

  B.begin (); // start transaction
              // automatically nested if one
              // is already running

  if (setlock(new Lock(LockMode.WRITE) ==
                                  Lock.GRANTED)
  {
    userDetails = obj;

    if (B.commit()) // aborts if cannot commit
      result = true;
  }
  else
    B.abort();

  return result;
};

public boolean save_state
                        (OutputObjectState os);
public boolean restore_state
                        (InputObjectState os);

private UserDetails userDetails;
};
```

An example of the server code is shown below. Apart from declaring instances of the required objects and invoking the methods for transferring funds between accounts and depositing the cookie, the programmer need only start and terminate the transaction. The transaction system will guarantee the outcome even in the presence of failures.

```
{
  AtomicAction A = new AtomicAction();
  BankAccount B1 = new BankAccount(UserNumb);
  BankAccount B2 = new BankAccount(PaperNumb);
  Cookie C = new Cookie();

  A.begin();

  if (B1.debit(amount) && B2.credit(amount))
  {
    if (C.depositCookie(UserDetails))
      A.commit();
    else
      A.abort();
  }
  else
    A.abort();
}
```

Once the application has been constructed, we can decide on the configuration. The transactional object within the browser represents the cookie, which is initially empty. Upon successful completion of the transaction, the cookie will have been stored for future use. The requirements on concurrency control for the cookie are minimal since there will be no concurrent access by multiple users; therefore, the local non-persistent concurrency control implementation can be used. Since this implementation can be guaranteed to work under all SecurityManagers, we require no alternate.

Obviously we would like to store the cookie on the user's local disk. However, security restrictions imposed by the browser's SecurityManager (or by the user if digital signatures are being used) may prevent this. Thus, we require an alternate form of persistence in these situations. In this example we shall assume that the newspaper site will provide a persistence service implementation which is available remotely should the local implementation fail.

After identifying the application configuration, we can construct the HTML document containing the configuration information which will be downloaded with the Java application. (The '~' and '!' characters preceding each attribute value are used for runtime type checking by `ObjectName`.) Importantly, there are no requirements from the application user: all implementations will be loaded across the network when required.

```
<HTML>
<HEAD><TITLE>Example Applet</TITLE></HEAD>
<BODY>
<APPLET CODE=TranApplet.class WIDTH=400
HEIGHT=200>
<PARAM NAME=OSClassName1
VALUE="~LocalObjectStoreImple">
<PARAM NAME=OSLocation1
VALUE="!/tmp/ObjectStore">
<PARAM NAME=OSClassName2
VALUE="~RemoteObjectStoreImple">
<PARAM NAME=OSLocation2
VALUE="!glororan.ncl.ac.uk">
<PARAM NAME=CCClassName1
VALUE="~LocalCCImple">
</APPLET>
</BODY>
</HTML>
```

The preferred type of the persistence service is LocalObjectStoreImple, with the attribute name OSClassName, and the location of the object store is the directory /tmp/ObjectStore. If this fails, the interface can use the alternate implementation RemoteObjectStoreImple which is on the specified machine. The concurrency service is local. If the programmer wishes to change the configuration of the application, only modifications to the HTML document are required.

7. Comparisons with other systems

We are not aware of any other working OTS/JTS compliant, configurable transaction system; therefore, in this section we briefly describe some systems which offer limited functionality.

7.1 Transactions through cgi-scripts

Figure 7 shows how it is possible to use cgi-scripts to allow users to make use of applications which

manipulate atomic resources [TRA96]: the user selects a URL which references a cgi-script on a Web server (message 1), which then performs the action and returns a response to the browser (message 2) *after* the action has completed. (Returning the message during the action is incorrect since the action may not be able to commit the changes.)

In a failure free environment, this mechanism works well, with atomic actions guaranteeing the consistency of the server application. However, in the presence of failures it is possible for message 2 to be lost between the server and the browser. If the transaction commits, the reply will be sent after the transaction has ended; therefore, other work performed within the transaction will have been made permanent. For some applications this may not be a problem, e.g., where the result is simply confirmation that the operation has been performed. If the result is a cookie, however, the loss of the cookie will leave the user without his purchase and money, and may require the service provider to perform complex procedures to verify the cookie was lost, invalidate it and issue another.

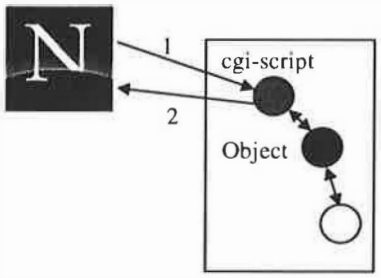


Figure 7: transactions through cgi-scripts

7.2 Transactions in persistent Java

There are several groups working on incorporating transactions into persistent Java [MA96]. These schemes are based on providing atomic actions with *orthogonal-persistence*: objects are written without requiring knowledge that they may be persistent or atomic: the Java runtime environment is modified to provide this functionality. The program simply starts and ends transactions, and every object which is manipulated within a transaction will automatically be made atomic. Although these approaches provide a convenient programming model, we believe that they are unsuitable for Web applications for the following reasons:

(i) They require changes to the Java interpreter and language. Applications written using these systems will only execute on specialised interpreters.

(ii) Both schemes assume that the entire application will be written in Java, and will not be distributed, i.e., it will either execute at the browser or at the Web server.

8. Concluding remarks

This paper has described the design and implementation of JTSArjuna, a standards compliant toolkit for the construction of fault-tolerant Web and Internet applications using atomic actions. The toolkit addresses the requirement for end-to-end transactional guarantees by allowing applications to be built which encompass Web browsers, rather than just Web servers. Transactional objects can reside within Web servers, and interact with objects and applications within other browsers or backoffice environments. As well as being standards compliant, the system does not compromise the security policy imposed at the browser's site. This means that applications can be built without requiring specific security policies, such as being able to write to the local disk. An application can be configured at build-time or run-time to adapt to the environment/user in which it runs, enabling the same application to execute anywhere.

Acknowledgements

The work reported here has been supported in part by a grant from UK Engineering and Physical Sciences Research Council (grant no. GR/L 73708).

References

[DF97] D. Flanagan, "Java in a Nutshell 2nd Edition", O'Reilly and Associates, Inc., 1996.

[GDP95] "The Design and Implementation of Arjuna", G.D. Parrington et al, USENIX Computing Systems Journal, Vol. 8., No. 3, Summer 1995, pp. 253-306.

[JSF95] J. S. Fritzinger and M. Mueller, "Java Security", Sun Microsystems, 1995.

[MA96] "Draft Pjava Design 1.2", M. Atkinson et al, Department of Computing Science, University of Glasgow, January 1996.

[MCL97] M. C. Little and S. K. Shrivastava, "Distributed Transactions in Java", Proceedings of the 7th International Workshop on High Performance Transaction Systems, September 1997, pp. 151-155.

[OMG95] "CORBAservices: Common Object Services Specification", OMG Document Number 95-3-31, March 1995.

[SKS95] S. K. Shrivastava, "Lessons learned from building and using the Arjuna distributed programming system," International Workshop on Distributed Computing Systems: Theory meets Practice, Dagsthul, September 1994, LNCS 938, Springer-Verlag, July 1995.

[SMW96] "The Design and Implementation of a Framework for Configurable Software", S. M. Wheater and M. C. Little, Proceedings of the 3rd International Workshop on Configurable Distributed Systems, May 1996, pp. 136-143.

[TRA96] "Transarc DE-Light Web Client Technical Description", Transarc Corporation, February 1996.

[VM96] "JTS: A Java Transaction Service API", V. Matena and R. Cattell, Sun Microsystems, December 1996.

Availability

Further information about JTSArjuna, including how to obtain and license the software, can be obtained from our Web site (http://arjuna.ncl.ac.uk) or by emailing M.C.Little@ncl.ac.uk

Secure Delegation for Distributed Object Environments

Nataraj Nagaratnam, *Department of ECS, Syracuse University*
Doug Lea, *Department of Computer Science, SUNY Oswego*
email: nataraj@cat.syr.edu, dl@cs.oswego.edu

Abstract

SDM is a Secure Delegation Model for Java-based distributed object environments. SDM extends current Java security features to support secure remote method invocations that may involve chains of delegated calls across distributed objects. The framework supports a control API for application developers to specify mechanisms and security policies surrounding simple or cascaded delegation. Delegation may also be disabled and optionally revoked. These policies may be controlled explicitly in application code, or implicitly via administrative tools.

1 Introduction

Open distributed computing environments must address four symmetrical security issues:

Services need not trust Users. For example, a database service may require that only certain users be able to modify records.

Users need not trust Services. For example, a person using an unknown word-processor application may not wish it to delete existing files.

Users need not trust Users. For example, a system administrator may only transiently allow an ordinary user to access a resource such as a tape drive.

Services need not trust Services. For example, a distributed database service may limit rights of different application programs that use it.

This paper describes the delegation-based mechanisms that underly a proposed framework, the *Secure Delegation Model.* SDM integrates support for these different aspects of security in Java-based distributed systems.

SDM is an architectural framework for structuring remote method invocations (RMI) among distributed components. It does not involve new encryption techniques, authentication protocols, or language constructs. SDM instead builds upon existing mechanisms, mainly those already established in the Java JDK1.2 security framework, to establish a practical basis for constructing flexible yet secure components and support infrastructure.

This paper focuses on the way in which delegation is structured and used in SDM to support secure operation when multiple components together provide a given service. Other aspects of the framework are described only briefly. Readers may find further details in [6].

The remainder of this paper is structured as follows. Section 2 defines Java-based security concepts and terminology surrounding Principals, Permissions, Privileges, Roles, and Security Domains. Section 3 introduces the SDM delegation framework. Section 4 describes the details of the resulting protocols, which are extended in Section 5 to handle dynamic revocation of delegated privileges. Section 6 briefly compares SDM to other approaches.

2 Concepts and Terminology

Principals. All parties associated with secure computation in SDM are known via *principals*: identities (unique names) that can be authenticated. We further restrict attention to *scoped* principals, for example *Syracuse's Nataraj*, where the scope represents an organizational domain (which may in turn be further

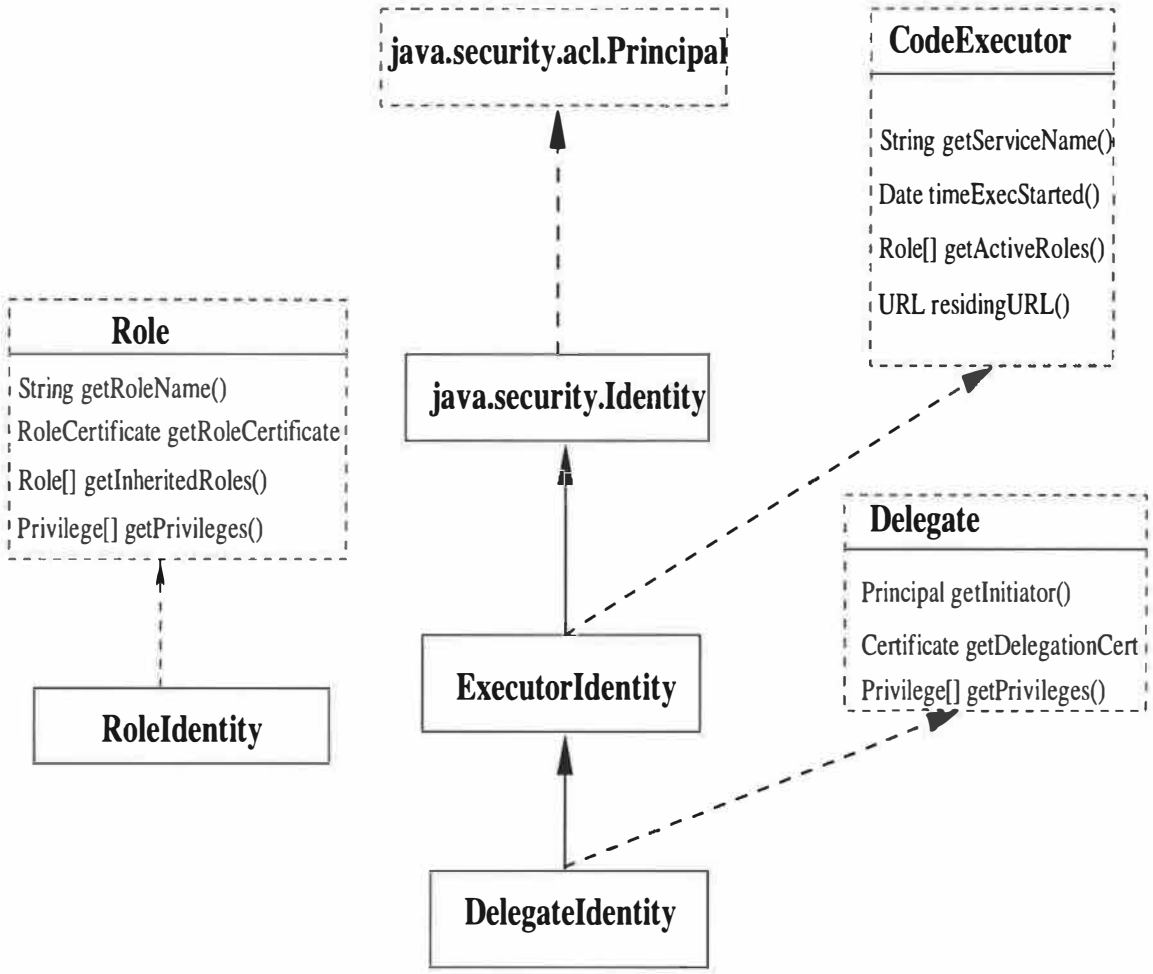


Figure 1: Principals in SDM

structured and scoped in any fashion). Principals are most often associated with individual people. However, they may also be associated with entities such as departments (as in *Acme's MarketingDept*, entire companies, or any other authenticatable unit. In SDM, we further categorize principals in terms of the properties and usages as discussed in the remainder of this paper and implemented via the classes and interfaces illustrated in Figure 1.

Signing. Java software components may be *signed*. The *CodeSource* associated with a component (i.e., one or more related classes) includes a set of signers recording the principals who developed that piece of code, or those who authorize the validity of the code. In the current Java model, a CodeSource encapsulates a set of signers who signed the class files, and the URL representing the location from which those class files are to be downloaded. Access can then be controlled based on such a CodeSource.

CodeExecutors. A signed component may be obtained from a software vendor and then executed used by a variety of users. Normally, the principal executing the code is different from the one that signed the classes. To clarify the resulting distinctions, we introduce the the concept of a *CodeExecutor* to be the principal invoking a given service, and upon which authentication, delegation or access control can be based.

Permissions. A *permission* is a named value conferring the ability (or formal consent) to perform actions in a system. We focus

mainly on permissions based on Access control policies, that grant permissions to principals on the basis of security attributes or privileges typically maintained via Access Control Lists (ACLs). In order to make an control decision, access decision functions compare the permissions granted to a principal against the permissions required to perform an operation. For example, permission to read a file `/tmp/foo.txt` can be denoted as `FilePermission: read:/tmp/foo.txt`.

Privileges. A *privilege* is a security attribute which may be shared by possibly many principals. We focus on the kinds of privileges defined in the XGSS and CORBA specifications, that include groups, roles, clearances and capabilities [7]. For example, Bill Clinton might have the privileges:

role: President-of-USA
capability: OccupyWhiteHouse
group: AmericanPresidents
accessId: WilliamClinton

Note that the permission to occupy the White House may be a capability transiently issued to him, with an expiration at the end of his presidency.

2.1 Roles

A given person or principal need not always have the same set of privileges. Rather than continually change them across different contexts, it is convenient to introduce the notion of a *role*, a set of actions and responsibilities associated with a particular activity [11] that might be adopted by any principal. A role is normally represented as a set of privilege attributes that a principal or set of principals can exercise within a context of an organization. The notion of a role does not add any power to a security framework, but instead improves manageability by adding an optional level of indirection. Role-based access control provides a higher level of granularity than approaches limited only to individuals. Because roles make transient privilege assignment much easier to administer, they have been widely adopted in security frameworks.

Role Certificates. A *Role Certificate* is an authenticatable device that provides evidence that a given principal possesses the attributes of a given role. In SDM, an executing Identity adopting a role is represented as a RoleIdentity. A RoleIdentity contains a RoleCertificate within it that it can be presented to any server. RoleCertificates have associated names and privileges, along with any other role hierarchy information; for example rules stating that all Managers are also Employees. When a principal authenticates itself and presents a valid role certificate, the privileges associated with that role becomes effective for the principal.

Adopting Roles. Roles may be used to obtain both extensions and reductions of privileges[1]. Reductions are typically performed in accord with a "least privilege" policy in which principals have only the privileges they need to accomplish a given task. Examples include:

- An administrator may want to have the powers of ordinary users most of the time, except when performing installation or user account creation.
- Users invoking untrusted software might want to reduce their powers before doing so.
- Users wishing to delegate only some of their privileges to others.

A principal A may adopt role R and act with the identity (**A as R**) when transiently obtaining or reducing powers. The privileges associated with a role work in the same way as those associated with principals. For example, a Manager role might have privileges:

group: CEOAnnouncementRecipients
group: companyBudgetReviewers
capability: MakeAppointmentOffer
-grantedBy Company
capability: ChargeCompanyCreditCard
-grantedBy Company

A principal plays a role by associating itself with one of its roles for a particular period of time. Thus, these privilege attributes must become associated with the principal. In SDM, this is accomplished by querying the RoleIdentity for its privileges.

Multiple Roles. A given principal can play multiple roles at the same time. So long as those selected roles are allowed to co-exist (i.e., they are not mutually disjoint roles), the principal can exercise the roles simultaneously, and thus obtain the union of privileges associated with them. To extend the above example, in a company intranet environment, access to a budget information file might be limited to the group named companyBudgetReviewers. A principal who has been assigned role of a Manager can access this information , due its privilege which contains the group membership. This group membership need not be explicitly assigned to the identity, but can just be associated with a role, in this case Manager. Similarly, the capability to make an offer to a candidate is automatic for a Manager as it contains the capability MakeAppointmentOffer having been granted by the company itself.

2.2 Domains

Protection Domains. A *protection domain* is an administrative scoping construct for establishing system and service security policies. The Java 1.2 security architecture provides support for protection domains and domain based access control. Currently, the creation of domains is based on a CodeSource indicating a URL and code signers. SDM extends this framework to include explicit support for principals.

Principal Domains. In SDM, each service is run on behalf of some principal, the CodeExecutor, who takes the responsibility for that service. In particular, given a remote service running on a machine at a port (mapping to a URL), there is an authoritative CodeExecutor responsible for that service. Implementation of SDM requires that the JDK1.2 domain model be extended to include principals, so that each CodeSource will also have a principal associated with it. One domain will be formed for each such <*CodeExecutor, CodeSource*>. Further authentication and access control (and delegation) may then be based on the CodeExecutor.

To support PrincipalDomains, the Java runtime system must maintain a mapping from <*CodeSource, CodeExecutor*> pair to their protection domains and also the mapping between protection domains and their privileges. This could, for example, be implemented at the execution stack level with the aid of class blocks and the executing environment frame, as illustrated in Figure2. More complete details can be found in [6].

3 Delegation

Secure delegation occurs when one object (the delegator or initiator) authorizes another object (the delegate) to perform some task using (some of) the rights of the delegator. The authorization lasts until some target object (end-point) provides the service. The essence of secure delegation is to be able to verify that an object that claims to be acting on another's behalf, is indeed authorized to act on its behalf[14].

The problem becomes more complicated in practice when we consider mobile objects, agents and downloadable content being passed around an open network, where the initiator need not have a clue of where all its representative objects are passed around. Additionally, a number of practical issues must be solved: The framework must be scalable in wide area networks, remain efficient under widespread use, and remain secure when dealing with complex trust relationships that can emerge in practice. Toward these ends, SDM provides a multifaceted approach, supporting any of several styles and protocols, including both simple (impersonation) and cascaded (chained) delegation, as well as means to disable and revoke delegation.

3.1 Protection Domains and Delegation

In Java (as of release 1.2), a protection domain is created for each CodeSource. In SDM, this notion is extended to form *PrincipalDomains* based on CodeExecutors as well. A target (or intermediate) controls access to its methods based on protection domains, i.e., <*PrincipalDomain, ProtectionDomain*> pair. Access is then controlled via the permission associated with both the CodeExecutor and/or CodeSource.

SDM delegation protocols are based on the notion that when a client delegates its rights to one object in a domain (i.e., when it enables delegation before invoking on a target object), it effectively delegates its rights to all the ob-

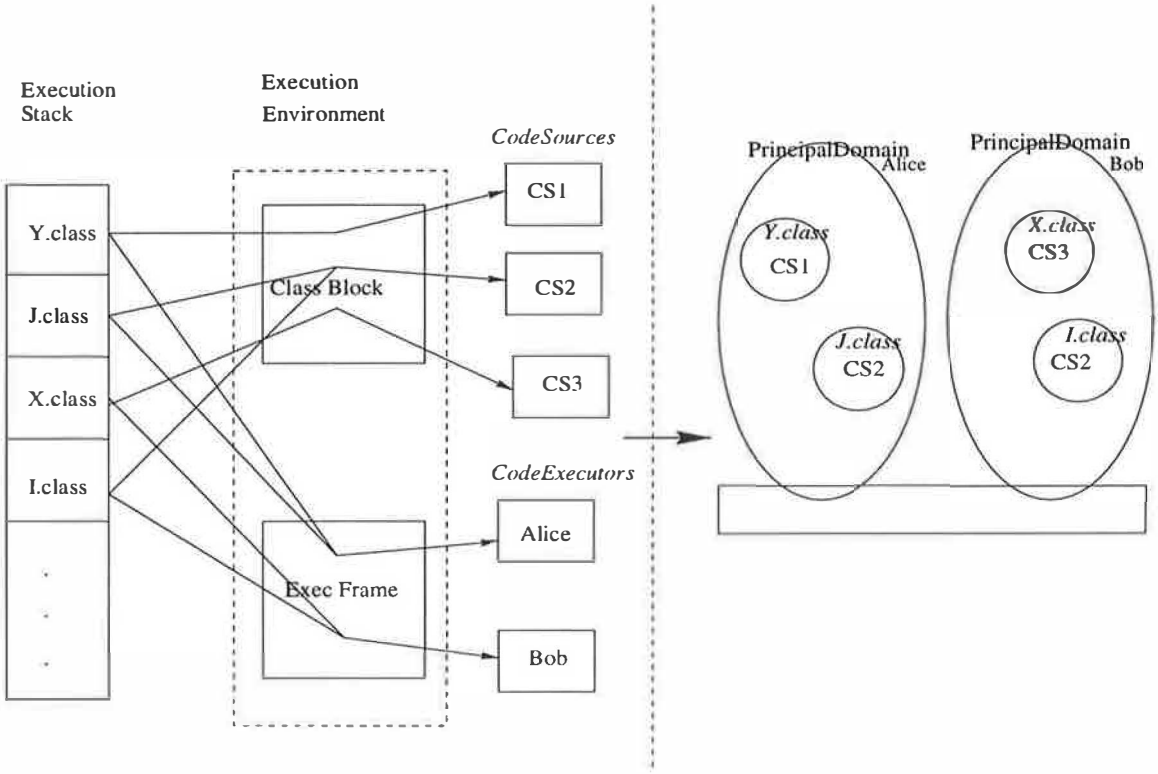


Figure 2: Obtaining domain information from execution stack

jects in that domain. This is implemented via *DelegationCertificates*, that behave analogously to RoleCertificates. In particular, a DelegationCertificate passed to a delegate can only be used by the object it is issued for.

A set of security *requirements* is associated with each object. If an intermediate object needs delegation from initiator, it *specifies* the delegation mode in its security requirements. Depending on the context (see Section 4), a delegation session may be established. If the target does not need to further delegate actions, no delegation certificate is generated by the client.

When initiating a delegation session, information about the initiating principal (CodeExecutor) is associated with the context of invocation. This is propagated through the underlying layer to the remote server (target) and gets associated (principal and CodeSource pair) with a protection domain. The target may provide access based on the identity of an individual or based on privileges it has (based on its effective role during invocation).

3.2 Modes and Chaining

A series of objects may be involved in a given service request. For example, suppose some object A (client) invokes a method on another object B (target). Object B might complete the task on its own or might in turn invoke a method on another object, C. In this context, object B which was earlier the target (for A's invocation) becomes a client for the method invocation on object C. Thus objects that are at first targets may later become clients. This effectively forms a *delegation chain* where object A is the *initiator*, object C is the *final target* and object B is an *intermediate*.

There are three different approaches, or *modes*, that may apply to such chains (see Figure 3):

NoDelegation. The intermediate exercises its own rights for further access.

SimpleDelegation. Impersonation; either restricted or unrestricted.

NoDelegation Only the invoking object's privileges are propogated

SimpleDelegation Only the initiator's privileges are propogated

CascadedDelegation Both initiator's and intermediate's privileges are combined

Figure 3: Delegation Chaining

CascadedDelegation. Combining rights of initiator and delegates.

After obtaining the delegation certificate from a delegator, an intermediate object might invoke a method on another object down the chain. At this point, the intermediate may decide to use only the delegator's privileges or combine it with its own privileges. This decision of either passing delegator's privileges only (impersonation) or combining its privileges too (composite) is based on the delegation mode specified for the intermediate object. Mode specification may be explicit through the application, or may be implicitly set by the administrator of that object service.

3.3 Controlling Delegation

Objects can explicitly enable delegation at the application level. This is accomplished by using an `AccessController` object. The `AccessController` method `enablePrivileged()` permits delegation. Method `enablePrivileged(RoleType)` is similar, except that when a role type is passed, the available privileges for that session are extended or restricted to the privileges associated with that enabled role. This functionality is not restricted to delegation. It can also be used whenever access to local methods and resources need special control. For example, consider a system administrator who logged in as a normal user but would like to exercise super-user privileges for an account creation. In this case, the administrator could invoke `enablePrivileged(superUser)` to enable super user privileges.

Either implicit or explicit enabling can be used to specify control in cases of Cascaded Delegation where the intermediary objects are *unaware* of secure delegation. If the intermediate is unaware, then the underlying security layer must effectively carry out either Simple Delegation or a special delegation mode set by

```
public class TravelAsst {
  :
  public void makeReservation() {
    :
    AccessController.enablePrivileged(managerRole);
    AccessController.enableSimpleDelegation();
    remoteAdmin.purchaseTicket();
    AccessController.disableDelegation();
    AccessController.disablePrivilege();
    :
  }
}
```

Figure 4: Sample Usage

an administrator. In SDM, explicitly specified modes are settable at the application level may and override the default mode set by the administrator. Either way, delegation requirements become attached to an intermediate object's reference. This set of requirements is made available to any client holding a reference to this remote (intermediate object) reference.

In contrast, a delegation-aware intermediate might explicitly enable delegation for a method call. In SDM, this explicit delegation may be performed at the application level. If delegation is enabled, the client may generate a delegation certificate and pass it on to the intermediate object. Otherwise, no delegation certificate is generated and the intermediate provides service using only its privileges and none of the delegator's (in which case, NoDelegation is the delegation mode).

An intermediate may also explicitly enable delegation using the `AccessController` methods `enableSimpleDelegation()` and `enableCascadedDelegation()`. The specified delegation mode is taken into account when privileges of the intermediate need to be presented to consecutive objects in the method invocation chain. Whether the intermediate's privileges are combined with the delegator's is based on the mode of delegation. The system can obtain the security requirements attached to any remote reference. The delegation, if required by the specified requirements (and target object is thus willing to act as a delegate), is activated appropriately from the context. Using the context of invocation, delegator's AccessControler determines the CodeExecutor who is executing client's code. This CodeExecutor becomes the Signer of a delegation certificate, and thus effectively the initiator of a delegation.

An example of application-level control is shown in the code segment in Figure 4. This code could be used to handle situations in which a client object invokes method `makeReservation()` on an object of type `TravelAsst`. The TravelAsst object might in turn invoke methods on a `remoteAdmin` object. In the sample code, the `travelAsst` explicitly enables delegation before further invocation on `remoteAdmin`.

3.4 Delegation Certificates

When an object decides to delegate a task to another object (effectively to the CodeExecutor of that object), it creates a delegation certificate. This certificate specifies the initiator, role it is delegating, any constraints that are bound to the delegation, a nonce, validity period and its DelegationServer name for handling queries regarding delegation revocation. A role certificate is associated with the role being delegated, which might contain a set of privileges associated with it.

A delegation certificate is generated using the CodeExecutor as FromPrincipal and the CodeExecutor of the remoteAdmin object as the ToPrincipal. Implementations could be based on public key cryptography using X.509 certificates, as illustrated in Figure 5. The associated role (and hence, set of privileges) is specified in the certificate.

A delegation certificate is issued for every

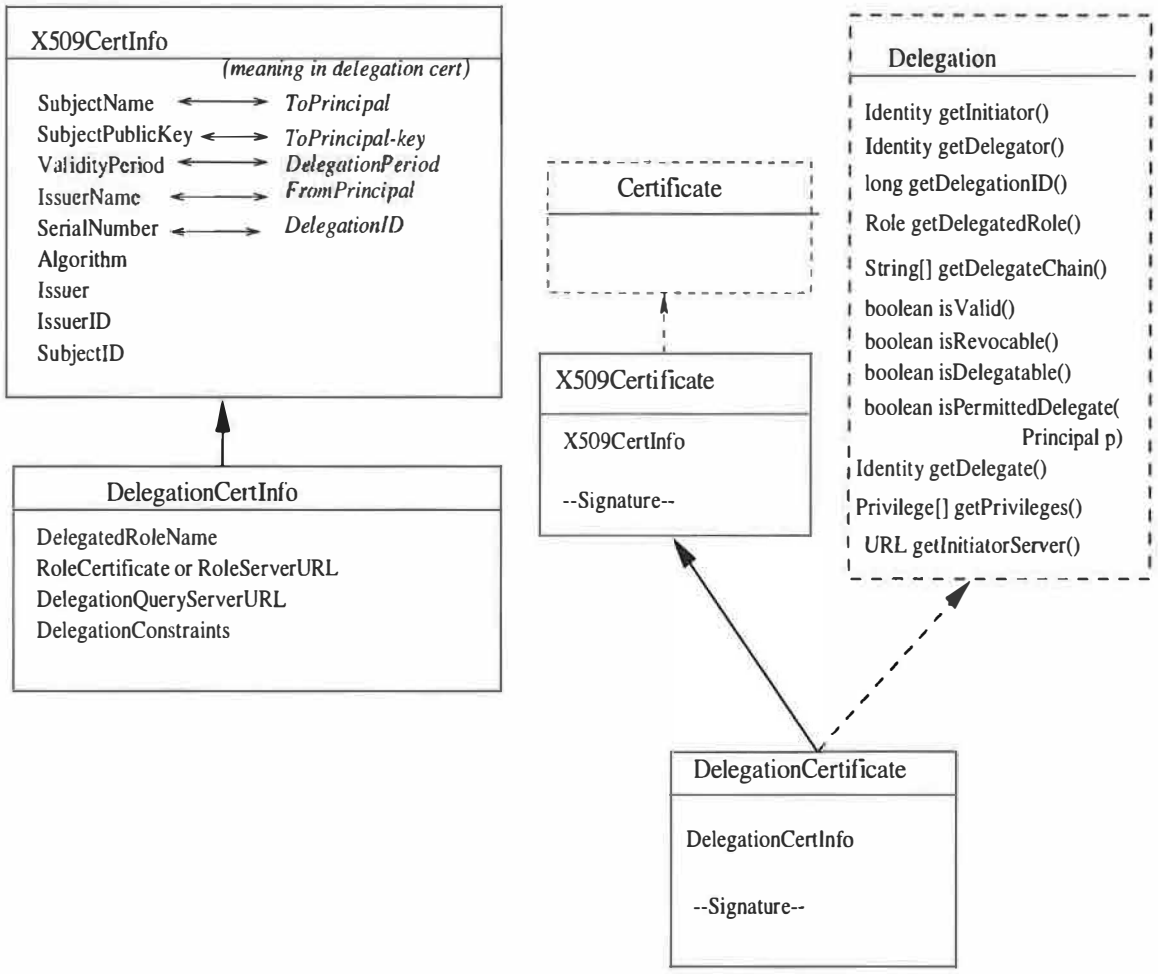


Figure 5: X.509 and Delegation Certificates

delegation session unless an earlier delegation has been set to remain valid for consecutive sessions. The type of the delegation certificate (SimpleDelegationCert or CascadedDelegationCert) reflects the kind of delegation that is activated for this session. If the delegation is revocable, the end-point makes sure that the delegation certificate is not revoked before it provides access.

Selection of consecutive delegates is made by an intermediate. Selected principal (CodeExecutor of the selected object for further delegation) is verified to be a permitted delegate by invoking the `isPermittedDelegate(Principal)` method on the certificate (DelegationCertificates must implement the `Delegation` interface shown in Figure 5). This method will scan through the list of exempted delegates (if any) and accordingly will return a boolean value, indicating whether or not the principal is a valid delegate.

4 Delegation Protocols

SDM employs a set of basic protocols that underly the usages described in Section 3. SDM delegation protocols specify what information gets exchanged when an object A invokes a method on object B. The underlying layer must determine the delegation mode to be enabled from the context and security requirements attached to the target (remote reference B). Thus, the security policy for an intermediate object governs which privileges and delegation mode to apply at any given context. (See Figure 6.)

Different rules apply for each of the combinations of required and specified modes that

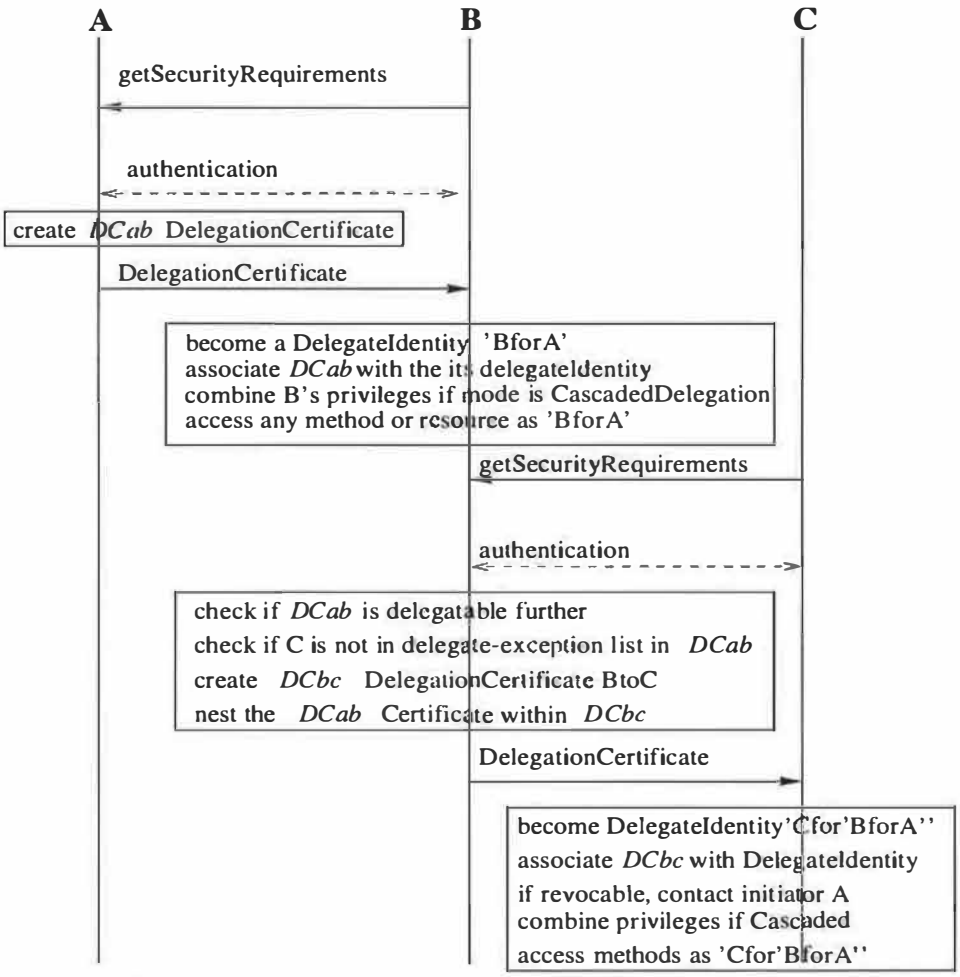


Figure 6: Main Delegation Protocol in SDM

can occur in a sequence of invocations from object A to object B to C. (i.e $A \rightarrow B \rightarrow C$):

does not enable Delegation, specifies NoDelegation. Delegation is disabled for this session. No delegation certificates are generated. Methods on object B are invoked as if invoked by object A, and methods invoked by object B on the next object in the delegation chain are invoked with object B's privileges and so on. Any object that is invoked by B will not get any information that reflects that A has delegated to B to complete the task.

does not enable Delegation, specifies Simple or Cascaded Delegation. Delegation is disabled for this session even though B requires it. When the operation that requires delegation from A to B is attempted, an exception is thrown and the operation is not carried out.

enables Delegation, requires NoDelegation. If the security requirements attached to B specify that delegation not be enabled for this session, then no delegation certificate is generated. The method on object B is invoked as if invoked by object A, and method invoked by object B on the next object in the delegation chain is invoked with object B's privileges and so on.

enables Delegation, requires Delegation. If B requires delegation, A must generate a delegation certificate, DC_{ab}, to B. This delegation certificate is available to B for any further invocation. Consider when B need to invoke a method on another object C. Such invocation on object C is carried out by B as

a DelegateIdentity **(B for A)** (i.e., B is a delegate and A is the initiator).

enables Delegation, specifies Simple Delegation. Further invocations (and delegations, if any) made by B are made as **(B for A)** with only the privileges of A being used. In other words, B impersonates A **(B as A)**. Any target that receives a request from B will authenticate B and obtain the delegation certificate DC_{ab}. Further control will be based on privileges of A and B's capacity to act as a delegate.

enables Delegation, specifies Cascaded Delegation. Further invocations and delegations are made by B by combining both the privileges of A (using delegation certificate DC_{ab}) and B (by providing necessary role certificates or identity certificates). In other words, B represents A by combining the privileges of both A and B. Any target receiving a request from B will base its access decision on A being a initiator and B being a delegate, with the combined privileges of both A and B.

4.1 Chained Invocations

Once the intermediate B has obtained the delegation certificate DC_{ab} from A, it has the authority to speak for A. To complete the service, B might have to invoke methods on other objects. When B selects C to be the next target, B represents an entity **(B for A)** and requires access to method invocation. At this point B exercises the type DelegateIdentity and during the process of becoming a DelegateIdentity, the delegation mode is considered to calculate the privileges of the delegate (here, B). In this case of B being a DelegateIdentity, B can authenticate for itself. Also, it provides the delegation certificate DC_{ab} to prove that A has indeed delegated the task to B. C authenticates that it is actually B it is talking to, through normal authentication procedures. It verifies the delegation certificate to be signed by A by verifying the digital signature of A that is engraved in the certificate DC_{ab}. These provide proof of the fact that "B speaks for A", abbreviated as or **(B for A)**[1]. The delegate's (B's) `getPrivileges()` method returns the privileges associated with B, which is either only the privileges gained through delegation (A's privileges only), or also includes the privileges of the delegate identity itself (both B's privileges and A's privileges). Thus the set of privileges returned by the delegate reflects the privilege of the identity **(B for A)** during that context.

Based on access control policies on the target C, the method invocation and any related resource accesses are controlled. These access control policies may be based on only the initiator (A) or might depend on the delegates as well.

If C requires delegation from its requester and B is a delegate for A possessing a delegate certificate DC_{ab}, i.e., **(B for A)** then:

1. B first checks if this delegation is forwardable, i.e., delegatable to further intermediaries.

2. If it is forwardable, B checks whether C is present in an exception list provided with the delegation certificate DC_{ab} (i.e, whether A disapproves any further delegation to C).

3. If delegation to C is not prohibited, B generates a delegation certificate DC_{bc} and passes on DC_{ab} with it to C. The delegation certificate DC_{bc} may contain: i) A's privileges only, ii) B's privileges only, or iii) a combination of both, depending on the delegation mode specified in the delegate identity B. Once B delegates to C, the delegation chain becomes $A \rightarrow B \rightarrow C$, i.e., **(C for (B for A))**.

In contrast, if B is not a delegate for A, that is if B had not specified delegation in its security requirements, then A would not have generated (and passed on) the delegation certificate DC_{ab} to B. In this case, when a request is issued to C, it is not possible for B to establish A as the original initiator due to the lack of a delegation certificate. So C must treat it as if the request originated from B and handle it accordingly, without having any idea about the involvement of A in the complete invocation chain. Extending this chain to one more principal, we get $A \rightarrow B \rightarrow C \rightarrow D$. If B does not require delegation and C does, then when the request reaches D, D will treat the

request to have initiated from B and delegated through C.

Thus, at any given time, control is based only on currently available information on the delegation chain and the specified modes and policies. SDM does not support any means of tracing back calls through intermediaries to obtain predecessor delegation certificates.

4.2 An Example

Consider an example of an user, A, using the services of a TravelAgent object, B. Let object B provide services related to travel reservations and travel arrangments. It might in turn need to make use of the services of AirlinesServices provided by an object, C. A obtains the reference of B and invokes the *makeReservation* method on B. Object B might specify, attached with its object reference, a set of security requirements. Let the security requirements specify that *Delegation* is required. In SDM, our system will analyse this security requirement attached to an intermediate object (in this case, object B) and whether A is willing to delegate (known from A's security specification attached to its object reference). Mapping this example to the delegation protocol described in Figure 6, the underlying system generates a delegation certificate and passes it on to B.

Let the travel agent B contact the airline object, C, to make an airline reservation by invoking the *purchaseTicket* method. At this point, B provides its certificates (preferred travel agent certificate, certified travel agent, etc) along with the delegation certificate issued by A. B acts as a delegate, acting on behalf of A, and makes a request to C. For this request B combines its own privileges (of being a preferred travel agent, authorizzation to make reservations, etc) along with the privileges of the intiator A (as the service might make use of A's credit card, or a travel coupon issued explicitly to A). Thus B makes use of CascadedDelegation facility provided by SDM while invoking the *purchaseTicket* method on the object C.

5 Revocation

Sometimes users and services need to *revoke* privilege assignments. Users change their minds; people leave groups, services change functionality, and so on. Even though it adds complexity, any practical delegation protocol must support revocation.

In SDM, revocability is an *optional* attribute of delegation. If performance is an issue, or revocation is somehow known to never ne necessary, the delegation can be made non-revocable. This facility to explicitly enable or disable revocation is again carried out using the AccessController object. The changed revocation status remains valid, until it gets changed again. The `AccessController` method `setRevocableDelegation(true)` enables delegation to be revocable until it is set otherwise.

If delegation is revocable, then the endpoint (but not necessarily any of the intermediate delegates) of a chain must be able to find out. In SDM, the DelegationID and delegation server (URL) associated with certificates define the uniqueness of a delegation certificate. If the endpoint has not seen the delegation certificate earlier, it must contact the DelegationServer of the initiator and verify its validity. And if it is not a one-shot delegation (a delegation that is valid for one access request only), the end point registers itself as a DelegationRevocationListener with the initiator.

When an end-point receives a service request from a principal, its AccessController checks if the service has been delegated through the invoking principal, and if so whether the delegation is revocable. If the delegation is not revocable, it goes ahead to provide/deny access according to the delegates privileges.

But if the delegation is revocable:

- The AccessController first checks if the delegation certificate is in a local <*delegationCertificate, status*> table.
- If the certificate is not present in the table, then this must be the first time this delegation certificate has been obtained. It contacts the DelegationServer of the initiator querying the status of the delegation.
- If the delegation is not one-shot, the user setting is analyzed to see if the change-

of-status notification is *periodic* or *aperiodic*.

- In the default aperiodic case, the end-point registers itself as a DelegationStatusListener with the delegation server for future updates only upon status changes.
- In the periodic case, the end-point registers itself as a DelegationStatusListener with the delegation server for future periodic updates at a given update interval.
- If the delegation is revoked (i.e., the delegation is no longer valid), the access is denied. Otherwise, the AccessController then provides or denies access according to resulting status and deletage privileges.

5.1 Revocation Notifications

In SDM, revocation is possible even when the initiator does not know the endpoint a-priori. When an end-point (final target) receives a revocable delegation request, it registers with the initiator as being interested in receiving revocation notifications. Thus each of such end-points register themselves as DelegationStatusListeners to the initiator. The initiator in turn maintains a list of end-points to whom its delegation has propagated. These end points will implement the NotificationHandler interface, to handle any event notification.

If the end point contacts the initiator every time before servicing a delegated request, then the end point is considered to follow a *pure pull* mechanism to obtain the status information from the initiator. A common alternative is *pure push* mechanisms, in which the initiator continually broadcasts out revocation information. Analyses of similar protocols using Broadcast Disks [3] show that pure pull provides extremely fast response time for a lightly loaded server, but as the server becomes loaded, its performance degrades, until it ultimately stabilizes. The performance of pure push is independent of the number of clients listening to the broadcast. But if the number of interested clients (end points) is large, then its a wastage of resources to send irrelevant data. A more serious problem is that the servers might not deliver the specific data needed by clients in a timely fashion. One solution suggested by Zdonik [3], is to allow the clients to provide a profile of their interests to the servers.

In SDM, the clients are the end points who are interested in the revocation status of certain delegations (serviced by those end points). When an end-point receives a delegated request, the first time around it pulls information from the initiator about revocation status and at the same time registers itself to receive delegation-related events. The profiles of those end points of interest in SDM, are the details on whether they require periodic push or aperiodic push. A *aperiodic push* is event driven – a data transmission is triggered by an event such as data update (in SDM, it is a change delegation status). Hence, end points (and hence, the NotificationHandlers) are notified of any change in delegation or its privileges by the initiator (which might use a helper object that implements EventGenerator interface). A *periodic push* is performed according to some pre-arranged schedule. The end point, when it registers itself with the initiator, will specify the time interval of periodic updates (pushes). Hence, the initiator will push delegation details at specified time intervals to the registered end points. This leaves it to the end point to specify whether it needs aperiodic or periodic (if so, the necessary time interval) pushes. Thus the end point need not pull information after its initial "pull" as the initiator will "push" (revocation) data to registered listeners (end-points). Either periodic or aperiodic, this *pull-once-push-many* approach supports revocation where an end point receives revocation notifications from an initiator.

An end point will decide to specify its interest in periodic or aperiodic pushes from the initiator based on how critical the revocation affects its service and its resources. For example, if the end point is a TelephoneDirectory service then providing information to a requesting delegate (a secretary object) about a revoked number is not very crucial, as the delegate might not misuse the telephone number. In this case, periodic pushes from the service department is not necessary and the end point might settle in for aperiodic pushes only. On the other hand, if the requested service is providing classified information, then the end point needs to know the revocation of the delegate (for example, a secretary object) immediately. In this case, short-interval peri-

odic pushes from the service department may be selected. The load on the server to keep pushing revocation status of its delegates becomes worth its cost when compared to the risk involved in providing classified information to any revoked delegate.

6 Status and Future Work

This paper has focussed on the way in which delegation is structured and used in SDM to support secure operation when multiple components together provide a given service. SDM builds upon exisitng mechanisms, mainly those already established in the Java JDK1.2 security framework, to establish a practical basis for constructing flexible yet secure components and support infrastruture. SDM extends the JDK1.2 framework to include explicit support for principals. We have provided implementation strategy for SDM to be built over the JDK1.2 framework.

As outlined in section 2.2, implementation of SDM requires that the JDK1.2 domain model be extended to include principals, so that each CodeSource will also have a principal associated with it. One domain will be formed for each such <*CodeExecutor, CodeSource*>. Further authentication and access control (and delegation) may then be based on the CodeExecutor.

To support PrincipalDomains, the Java runtime system must maintain a mapping from <*CodeSource, CodeExecutor*> pair to their protection domains and also the mapping between protection domains and their privileges. This could, for example, be implemented at the execution stack level with the aid of class blocks and the executing environment frame, as illustrated in Figure2.

In future, we intend to implement our SDM delegation framework over the JDK1.2 security framework. We have already implemented access control mechanisms [16] based on CodeSource information. We plan to extend the mechanism to include the information on principals to further control any access requests.

7 Discussion

SDM provides a realistic security framework for Java-based distributed object systems. It isolates the complexities of the underlying protocols necessary to provide a very wide range of security policies and trust levels. It presents application writers and system administrators with a flexible, uniform API. SDM appears to be the most conservative extension of the Java 1.2 security architecture that simultaneously supports both delegation- and role-based security, along with revocation mechanisms that are often needed in practice.

The design of SDM has also benefited from other work in security architectures, but differs from previous systems in significant ways:

DSSA. Roles are not explicit in DSSA[4] and are achieved through their notion of groups, whereas explicit support for roles is provided in SDM. DSSA supports only combined delegation, whereas SDM supports both combined delegation and composite delegation.

Varadharajan et al. The main revocation strategy proposed by Varadharajan et al [14] propagates revocations through delegates. These revocations might not take effect due to network problems or other distributed failures. Another solution proposed in [14] assumes prior-known end point. This is also supported in SDM. Approaches suggested in their paper require changing the key associated with a principal. This is not effective in public key systems, which are generally more manageable and scalable in distributed system (and are supported in SDM). They also suggest passing a *read* capability of the delegation token and not the token itself. Our approach is vaguely similar in that the end point need to contact the initiator before servicing. But by using the *pull-once-push-many* approach, SDM does not need to contact initiator because the initiator will multicast revocation details, if needed.

SESAME. SDM provides both simple and cascaded (composite, combined) delegation with support for constraints whereas SESAME[8] supports only simple delegation. Also, unlike SESAME, SDM also supports scalable distributed naming schemes.

Kerberos. In Kerberos[13], the end-point contacts authentication server for every signature authentication as it uses shared key approach. SDM allows implementation via pub-

lic keys and hence need not contact an authentication server every time. Kerberos does not support roles. Principals can restrict their privileges before delegation. Also, kerberos does not support cascaded delegation. There is no mechanism mentioned for revocation.

Taos. Taos[15] has no mechanism for revocation implemented. It supports the notion of a Privileges Server. Every time an access is processed by the end-point, it contacts the privileges server to validate the certificate.

DCE. DCE[2] does not provide any facility for revocation. Also, DCE uses shared key authentication which is not as scalable in distributed environments.

7.1 Limitations

We are aware of the following limitations of SDM, that reflect some of engineering tradeoffs encountered in its design:

- SDM relies on initiators to enable delegation. If they do not, delegation will never be enabled and hence no delegation certificates will be generated. If delegation is not initially enabled, at a later stage during method execution (through delegation), a target object cannot determine the original initiator of the request. The only way to find out the original initiator would be to use a call back trace mechanism, which is not supported in SDM.

- SDM does not support any means to check whether a principal adopts mutually disjoint roles. SDM cannot ensure that roles adopted by a principal do not conflict (for example simultaneously requiring and prohibiting rights).

- Although the pull-once-push-many approach is an efficient approach, event notification does not carry any real-time guarantees due to possible network latency. Before a revocation event notification arrives to a listener, the listener might have already allowed the revoked delegation. Hence, the event notification across distributed systems in SDM is not atomic.

References

[1] M. Abadi et al, "A Calculus for Access Control in Distributed Systems", *ACM Transactions on Programming Languages and Systems,* pp 706-734 Sept 1993.

[2] M. Erdos, J. Pato, "Extending the OSF DCE Authorization System to Support Practical Delegation", *PSRG Workshop on Network and Distributed System Security,* Feb 1993.

[3] M. Franklin, and S. Zdonik, "A Framework for Scalable Dissemination-Based Systems", *Proceedings of OOPSLA '97,* pp Oct 1997.

[4] M. Gasser, E. McDermott, "An Architecture for Practical Delegation in a Distributed System", *Proceedings of the 1990 IEEE Symposium on Security and Privacy, p20-30,* May 1990.

[5] L. Gong, "Java Security Architecture (JDK1.2)," *Draft document, Revision 0.5, available from Java Developer Connection, http://java.sun.com* .

[6] N. Nagaratnam, "Secure Practical Delegation for Distributed Object Environments," PhD Dissertation, Syracuse University, 1997.

[7] Object Management Group, *CORBA: Security Service Specification*

[8] T. Parker, D. Pinkas, "Sesame V4 - Overview", *Issue 1,* Dec 1995.

[9] R. Sandhu et al., *Role-Based Access Control Models,* IEEE Computer

[10] R. Sandhu et al., *Role-Based Access Control,* Proc of 10th Annual Computer Security Applications Conference, 1994, pp 54-62.

[11] R. Sandhu, and P. Samarati, "Access Control: Principles and Practice", *IEEE Computer,* pp 40-48, Sept 1994.

[12] Karen Sollins, "Cascaded Authentication,", *Proceedings of the IEEE Symposium on Security and Privacy,* IEEE Computer Society, 1988, pp 156-163.

[13] J. Steiner et al, "Kerberos: An authentication service for open network systems", *Proceedings of the Usenix Winter Conference,* 1988, pp 191-202.

[14] Varadharajan et al., " An Analysis of the Proxy Problem in Distributed Systems", *Proceedings of the IEEE Symposium on Security and Privacy,* 1991, pp 255-275.

[15] Wobber, Abadi, Burrows, and Lampson, "Authentication in Taos Operating System", *ACM Transactions on Computer Systems, Vol. 12, No. 1,* February 1994.

[16] N. Nagaratnam, and S. B. Byrne, "Resource Access Control for an Internet User Agent", *Proceedings of the USENIX Conference on Object Oriented Technologies and Systems '97,* June 1997.

[17] A. Wollrath, R. Riggs, and J. Waldo, "A Distributed Object Model for the Java System," *Proceedings of the USENIX Conference on Object-Oriented Technology and Systems '96,* July 1996.

COBEA: A CORBA-Based Event Architecture

Chaoying Ma and Jean Bacon
University of Cambridge Computer Laboratory
Pembroke Street, Cambridge CB2 3QG, United Kingdom

http://www.cl.cam.ac.uk/Research/SRG/opera/
cm@cl.cam.ac.uk, jmb@cl.cam.ac.uk

Abstract

Events are an emerging paradigm for composing applications in an open, heterogeneous distributed world. In Cambridge we have developed scalable event handling based on a *publish-register-notify* model with event object classes and server-side filtering based on parameter templates. After experience in using this approach in a home-built RPC system we have extended CORBA, an open standard for distributed object computing, to handle events in this way.

In this paper, we present the design of COBEA - a COrba-Based Event Architecture. A service that is the source of (parameterised) events *publishes* in a Trader the events it is prepared to notify, along with its normal interface specification. For scalability, a client must register interest (by invoking a *register* method with appropriate parameters or wild cards) at the service, at which point an access control check is carried out. Subsequently, whenever a matching event occurs, the client is *notified*.

We outline the requirements on the COBEA architecture, then describe its components and their interfaces. The design and implementation aim to support easy construction of applications by using COBEA components. The components include event primitives, an event mediator and a composite event service; each features well-defined interfaces and semantics for event registration, notification and filtering. We demonstrate that COBEA is flexible in supporting various application scenarios yet handles efficiently the most common event communications. The performance of server-side filtering for various registration scenarios is presented. Our initial experience with applications is also described.

1 Introduction

Event communications are asynchronous compared with the request/response operations in the standard client/server model for distributed systems. There are many application areas where event-driven operation is the most natural paradigm: interactive multimedia presentation support; telecommunications fault management; credit card fraud; disaster simulation and analysis; mobile programming environments; location-oriented applications, and so on. An event is defined as the occurrence of some interaction point between two computational objects in a system. Such a point may reflect an internal change of state of the system, or an external change captured by the system. An event can be a base event which has a single source of generation, or a composite event which correlates multiple base event occurrences to be signalled as a whole. Events may be pushed by suppliers to consumers (the push model) or pulled by consumers from suppliers (the pull model) through specific or generic interfaces; such communication may be direct, or indirect i.e. through an intermediate object between the consumer and the supplier. Active systems monitor the occurrences of events and push them through to client applications to trigger actions [2, 5]. In contrast, passive systems require client applications to poll to detect event occurrences. An active system is therefore inherently more scalable than a passive system.

For example, in an Interactive Multimedia Presentation support platform [3] a script specifies *event-condition-action* rules to drive the interactive presentation. The events "Roger appears" and "Roger disappears" may be associated with frames 2056 and 3092 of a video presentation. If the user clicks on Roger (an area of the screen marked during pre-

processing of the film) after "Roger appears" and before "Roger disappears" then the pause method is invoked on the video, a new window pops up, text on Roger is displayed and the film resumes when the user clicks again. Location devices, such as active badges or electronic tags, are another source of events. We may wish to analyse how users behaved during a fire drill in order to determine bottlenecks in a building [2]. We may arrange for our programming environment to move with us when we are detected moving from one workstation to another [1]. In telecommunication, various events are monitored by the management system and used for network analysis and fault recovery [12].

We have designed an architecture for building active, event-driven systems in a large distributed environment, where there is potential for high volumes of event traffic; for instance in telecommunication applications, a single source can generate tens or hundreds events per second. The design focuses on providing components to support easy construction of applications. The components include event primitives, the mediator and the composite event service; each features well-defined interfaces and semantics for event registration, notification and fine-grain filtering. We demonstrate that COBEA is flexible in supporting various application scenarios yet handles efficiently high event volumes in the prototype implementation. After experience in using this approach in a home-built RPC system we have extended CORBA, an open standard for distributed object computing, to handle events in this way.

1.1 Existing Work

Architectural frameworks for event handling in large distributed systems are discussed in [2, 7, 14, 19, 23, 24, 25]. The CORBA Event Service [14] introduces the concepts of event channel, supplier and consumer. An event channel is an intermediate object which decouples the supplier and the consumer. Event communication may be untyped in which a single parameter of type "any" is used for passing events; applications can cast any type of data into this parameter. For typed event communication, an interface **I** is defined in CORBA IDL. In the typed push model, suppliers invoke operations at the consumers using the mutually agreed interface **I**; in the typed pull model, consumers invoke operations at suppliers, requesting events, using the mutually agreed interface **Pull<I>**. Some application scenarios may be supported by the push and pull models, or a combination of them. But it is not possible for a client to select only those events which are of interest, at fine granularity, by means of a detailed specification of parameters and wild cards. Furthermore, in the push model, the supplier is responsible for acquiring the reference to an appropriate notification interface in order to push events to the consumer. The CORBA event service also lacks the ability to filter events; only filtering by interface type is available. Other major limitations include overly general, thus inefficient for many applications, lack of standard semantics and protocols for event channels and lack of type safety in untyped interfaces. Schmidt and Vinoski have reviewed the CORBA event service [21].

The Cambridge Event Paradigm [2] addresses some of the shortcomings mentioned above as well as some advanced event handling issues. The *publish-register-notify* mode is very well supported: a service that is the source of (parameterised) events publishes in a Trader the events it is prepared to notify, along with its normal interface specification. For scalability, a client must register interest (by invoking a register method with appropriate parameters or wild cards) at the service, at which point an access control check is carried out. Subsequently, whenever a matching event occurs the client is notified. Filtering by parameters including wildcard parameters and by event types at event sources are the key features, which eliminate the need of placing filters between the event server and client. For example, users may specify filtering criteria which describe the **PrintFinished** event on a file named "foo" (by giving the file identifier upon event registration), or on every file (by giving a wildcard file identifier "*"). The Cambridge work focuses on providing event handling primitives which are based on the direct push model. A heartbeat protocol has been incorporated, which can be tuned for the trade-off of computation cost between timely and delayed event evaluation in order to ensure correctness in the light of network failures. Access control on event registration have also been proposed; you may not be allowed to monitor the movement of your boss, for instance. In addition, a Composite Event Language and an evaluation engine have been developed to allow the use of composite events. The language currently has five operators: **Without (-)**; **Sequence (;)**; **And (&)**, **Or (|)**; **Whenever ($)** (see [8] for details). For example, a composite event **$Enter(x, r)** will trigger whenever someone **x** enters a room **r**; and an event **Enter(a, 123)** |

Enter(b, 123) will trigger if either person **a** or **b** enters room **123**.

The design, however, is based on a conventional RPC system rather than an object-oriented paradigm. The implementation uses MS-RPC3, a locally developed RPC system, thus has limited interoperability. Furthermore, it requires extension to the MS-RPC IDL for specification of events, thus has the need to marshal events separately from ordinary RPCs. It does not directly address issues of indirect event communications, although a composite event server may be used as an event mediator.

1.2 Objectives

Motivated by observation of the shortcomings of the existing work and the emerging new requirements, in particular from the application domains such as telecommunications, we have designed COBEA for event handling to extend the existing Cambridge Event Paradigm and the CORBA event service. The main goal is to design an architecture which provides a framework for object-oriented design and development of active application systems, especially in a large distributed environment. COBEA extends the CORBA Event Service, namely by supporting the *publish-register-notify* mode, parameterised filtering, fault-tolerance, access control, and composite events; all these features are missing from the CORBA event service. COBEA is a reincarnation of the Cambridge Event Paradigm with all the features mentioned above as well as support for dynamic addition of new event types and event mediator.

The basic requirement on an architecture for event handling is that events can be identified, classified, detected, specified and asynchronously reported to any interested party through a standard or application-defined interface. A general architecture, based on which large-scale distributed active application systems can easily be constructed, is clearly required. We believe that both direct and indirect event communication should be supported based on a wide range of application requirements, e.g. in the areas of CSCW (Computer Supported Cooperative Work), management in network, telecommunication and distributed systems, multimedia systems and mobile systems [2, 3, 6, 9, 13]. We focus on supporting the push model, i.e. after explicit registration of interest by consumers, events are pushed directly or indirectly by suppliers to consumers (a.k.a. the Notification Model). Event registration is essential for receiving selective event notifications. Such a scheme not only solves the main problems with synchronous communications - the saturation of network resources caused by polling operations, but also solves the problem of end user saturation caused by pushing everything through. The pull model can easily be supported by using existing technologies thus specific support is not necessary. The goals of COBEA are, in outline, as follows:

- Support direct/indirect notification of events
- Support interfaces for event notification
- Support interfaces for event registration
- Support fine-grain event filtering
- Support interfaces for management (e.g. register suppliers with an event mediator)
- Support composite events
- Support dynamic addition of user-defined event types
- Support security on event accesses (e.g. role-based event accesses)
- Support Quality of Service (e.g. reliable or fast, priority-based event delivery)

2 Overview of COBEA

The components of COBEA include event handling primitives with which an event sink and an event source interface are defined, and event services namely a mediator and a composite event server. The architecture is illustrated in Figure 1, where components may be as primitives (grey circles) or stand-alone servers (white circles). These components can be used by applications via standard interfaces; and once included, they handle events for the applications as indicated by the arrows in dotted lines. Figure 2 shows inheritance structure of the component interfaces, in which an application may define its own typed interfaces for handling events with a choice to extend or not to extend the standard/generic interfaces defined in COBEA. We focus on the direct/indirect push model which forms

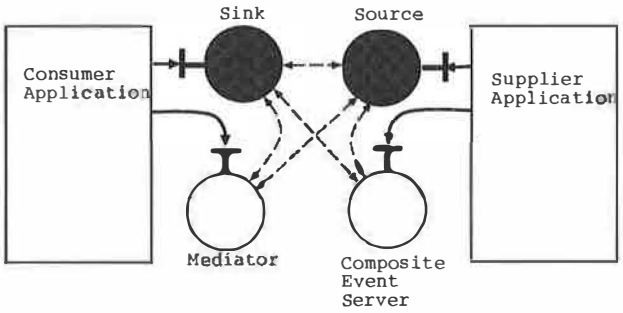


Figure 1: The COBEA Architecture

the core of event communication. The pull model can easily be supported by using the CORBA request/response, or most RPC communication primitives, or the forthcoming CORBA Messaging Service [17], or by incorporating the pull interfaces defined in the CORBA event service.

The event primitives support mainly event registration and notification operations defined by the event sink and event source interfaces. An application object can play the role of an event consumer or supplier by supporting the event sink or source interface respectively, or by supporting both interfaces, while providing other services at the same time (Figure 3 shows the incorporation of the event primitives in an application). Once registered the interest, the consumer will be notified whenever the event occurs. New interfaces for application-specific event handling can be derived from the primitive interfaces.

The event services define a number of objects acting either as event mediators or providing services for handling composite events. The main task of a mediator is to decouple the consumer and supplier by accepting events from the suppliers, and passing events only to the interested consumers. Thus consumers and suppliers do not need to know each other for communicating events. Many applications require notification of events from a number of suppliers in a specified pattern of combined events from these different sources. A composite event service is designed to meet such requirements.

One design principle is that the architecture should be lightweight yet powerful enough in order to support the construction of various distributed active systems. Some of the interfaces in COBEA may be defined by inheriting from the CORBA event service interfaces; for instance, the **snk** interface can extend

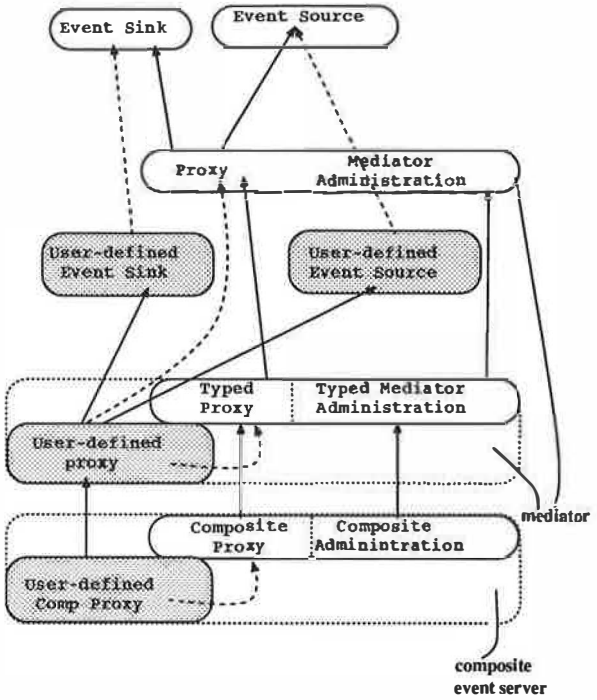


Figure 2: The Inheritance Hierarchy

Legends: round-corner rectangles represent objects; some object such as a mediator supports two interfaces. Shade objects have user-defined interfaces; other objects support standard/generic interfaces. Arrows show the inheritance from the interface that they point to; applications may choose whether to inherit if the arrows are in dotted lines.

the CORBA **PushConsumer** interface. Extending the CORBA event service this way means, however, that all the interfaces defined by the CORBA event service must be supported. We believe it is not necessary because in our Notification Model, most CORBA event service interfaces are undesired to use by applications. Later in Section 5.1, we will discuss how COBEA can be made to work with the CORBA event service.

Another design principle is that a filter should normally be placed on a supplier or a server to reduce the traffic to the consumers; the filtering criteria can be checked either at the supplier or at the server. It is important that filters should be kept as simple as possible. Sophisticated filtering which is less commonly used by most applications can be done at the application level rather than at the event system support level. There is a trade-off between the volume of event traffic generated and the complexity of supplier, mediator or consumer objects. Related work such as the ECA (Event-Condition-Action) rules in active databases [4, 5, 22] uses conditions

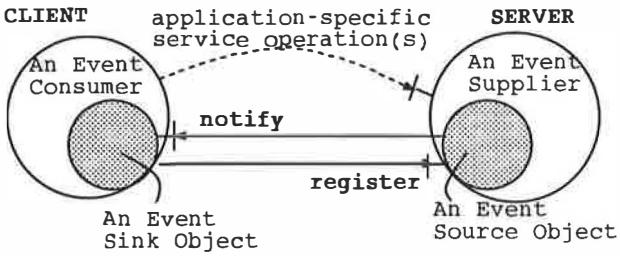


Figure 3: Direct event communication through primitives

which are like filtering criteria but can not be separated from either the evaluation engine or the action to take upon event occurrences. In our event architecture, filters can be placed at an event server, at a supplier, at a consumer, or chained among them, thus allowing less event traffic and greater flexibility.

Three options are available for implementing COBEA on top of the general-purpose communication system (e.g. an RPC system): create a new description language for specification of events, extend an existing IDL, or construct libraries to work with an existing IDL and its RPC system. The first two approaches allow the freedom to experiment with new ideas; the second approach in particular allows a standard extension to RPC systems for event specification. However, experience shows that application programmers are very reluctant to move existing programs, or write new ones, to make use of a non-standard environment. It is also very cumbersome for small research groups to maintain a non-standard RPC system, and keep its capabilities and performance competitive with that of a standard system. Thus, we base the implementation on CORBA - an open standard for distributed object computing [15]. We make the interfaces standard or follow a well-defined design pattern instead of using a non-standard IDL. As interoperability is concerned, CORBA 2.0 is designed to deal with heterogeneity and interoperability while most RPC systems are not.

This CORBA-based approach has the following advantages:

- Uses only standard IDL for events.
- No need to have a separate marshaling package for handling events.
- It is possible to allow the number of parameters in a registration interface to be different from that in the corresponding notification interface for the same type of event, if an application so requires.
- Type safety can be handled properly.

Based on the architectural framework, we are currently implementing a class library for COBEA. We have implemented the primitives, a composite event evaluation engine plus a parser based on the composite event algebra developed at the Cambridge Computer Laboratory. We are implementing two types of event service, namely an event mediator and a composite event service; the latter will incorporate the evaluation engine and the parser mentioned above. Furthermore, applications based on COBEA can easily be made to work with the CORBA Event Service, because all COBEA interfaces are specified using standard CORBA IDL. We have also developed a fault detection system for telecommunication network management based on COBEA.

3 The Design of COBEA

3.1 Primitives and Interfaces

Two objects are identified in the event notification model: a source and a sink of an event. It is essential for an event sink to receive events sent by an event source. The event source should support interfaces for event registration and deregistration; the event sink should support an interface for event notification. Both objects should also support a disconnect operation. The primitives also support the passing of a generic event header with standard attributes (properties), such as event identifier, creation time, type name, event source identifier and priority code. In addition, the interfaces allow a single parameter - event body - for passing application-specific event data dynamically. If more parameters are to be passed in applications, new interfaces should be defined, which can extend the primitive interfaces.

The standard interfaces are specified in the CORBA IDL as follows. The definition of exceptions is omitted.

```
module BaseEvent {
```

```
exception ...
struct EventTime
 {long sec; long usec; string clock_id;};

struct EventHeader {
 long id;  //the event id
 EventTime create_time;
 string event_type;
 string source_id;
 long priority; //severity code of event
 };

struct Duration {
 EventTime begin; EventTime end;
 };

struct ConsumerSpec {
 Object consumer_ref,
           //the consumer object reference
 Duration, //for time specific filtering
 string QoS, //QoS constraint
 string who, //for access control
 };

interface Snk {
 void notify(in EventHeader e, in any data)
 raises (NotConnected);
 void disconnect_snk();
 };

interface Src {
 void register(
  in EventHeader e,
  in string header_filter,
  in any event_body,
  in string body_filter,
  in ConsumerSpec consumer,
  out long uid, //the consumer id
  out long eid) //the registration id
 raises (RegistrationFailed);
 void deregister(in long uid, in long eid)
 raises (UserNotFound, EventNotFound);
 };
};
```

At registration of interest, a number of filtering parameters are allowed, including a duration for specifying the start and end time of events of interest. A filter each for the event header and the event body can also be specified, which is defined by a string containing operators including "* " for wildcard, "==", ">=", "<=", "<", ">", and "!=" for comparison. Filters can be used in combination with the given value of the parameters in the event header or body. The position of the operators in a filter is important; they correspond to the position of the parameters in the event header or body. Each operator is two characters long with a trailing space in case of ">", "<" or "*". For more complex filtering, see the section on the composite event service. In addition, the relation among the expression of the parameters is conjunction. The parameter **QoS** is for the consumer to specify quality-of-service requirements such as reliable delivery or fast (unreliable) delivery of events. The parameter "who" can be used to pass user identification for access control. Upon registration, a **Template** will be created which describes what event should be sent to which consumer.

For notification, an event matching the template of an event registration should be passed by the source to the registered sinks. Upon notification, actions can be taken at the consumer depending on applications.

An event communication can be broken by invoking a **disconnect_snk** operation at the event sink, or a deregister operation at the event source. A deregister operation will either remove a registered consumer with all related event templates or only a particular event template. If a communication is closed by the supplier, the consumer receives a notification through the **disconnect_snk** operation. The communication can only be resumed upon another **register** operation.

It should be noted that the defined interfaces allow only a standard event header with fixed number of parameters; the interfaces are also standard. For many applications, specific interfaces can be defined by following a well-defined design pattern in IDL files, e.g. **register<T>()**, where **T** may by substituted by a **DrawEvent** or **AccountingEvent** for a drawing or an accounting application respectively. A common set of event manipulation operations, such as comparison of the occurrence of an event against the registered templates, are supported through a common class library.

An example of an application-defined event sink interface may look as follows, where T_i is a type name. k parameters are used in this example. The interface extends to the standard **Snk** interface.

```
interface Snk<T>: BaseEvent::Snk {
 void notify<T>
  (in T1 arg1, in T2 arg2,..., in Tk argk)
 raises (NoSuchType, NotImplemented);
 };
```

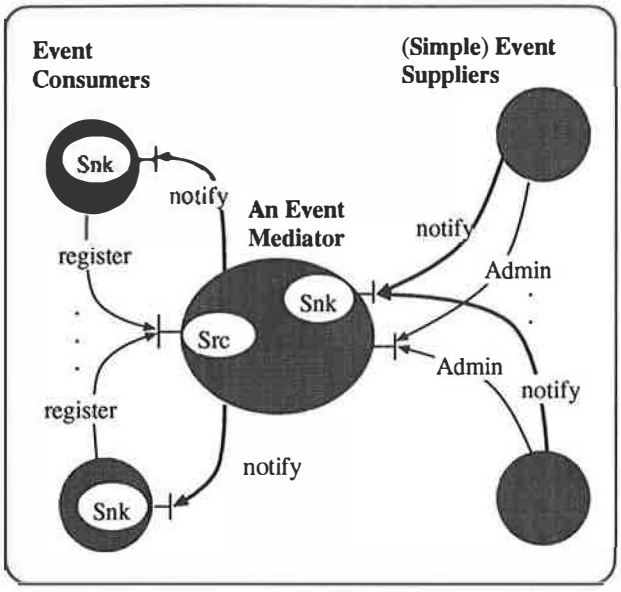


Figure 4: Indirect event communication through a mediator (only simple suppliers are shown)

3.2 Mediator and its Interfaces

For many applications, it is useful to have an event mediator. Figure 4 shows the indirect model of event communication. The advantages of having a mediator are: (1) a consumer or a supplier does not have to keep all the contacts to every event supplier or consumer but only the contact to the mediator; (2) simple event suppliers can be built which do not support a registration method; (3) commonly used filters may be built once for all; e.g. a filter at a mediator may be placed for all **faulty** events from one or more suppliers; (4) it is also easier to adopt group communication protocols such as a reliable multicast protocol at a mediator.

One assumption on the mediator is that a supplier needs a mediator to publish events; an event may be published by its type name and/or attributes (parameter names). A consumer needs to find a mediator to receive events. Finding a mediator is orthogonal to using it. Particular bindings between mediators, suppliers and consumers may also be arranged.

For standard events, the interfaces are as follows. Users may attach an application-specific piece of information when registering a new supplier.

```
module Mediator {
exception ...
interface Admin {
 Object new_supplier(
  in string appl_info,
  in boolean relay,
  out string uid)
 raises(RegistrationFailed);

 void remove_supplier(
  in string uid,
  in string application_info)
 raises (NoSuchSupplier, NoFound);
 };

interface proxy:
 BaseEvent::Snk, BaseEvent::Src {
 proxy lookup(in string type_id)
 raises (NotFound);
 };
};
```

For application-specific events, the interfaces should again be defined in the IDL files. Moreover, a generic interface is required for registration and notification of these events in many applications, e.g. event notification in telecommunication management. The generic interface allows dynamic addition of new event types and does not require all event types to be defined in IDL files. It is possible for a particular implementation to support only the generic interface. An exception **NotImplemented** will be raised if operations defined by **Snk<T>** or **Src<T>** are invoked. The generic interface of a mediator is defined as follows.

```
module TypedMediator {
exception ...
interface TypedAdmin: Mediator::Admin {
 Object new_typed_supplier(
  in string type_id,
  in boolean relay,
  out string uid)
 raises(RegistrationFailed);

 void remove_typed_supplier(
  in string type_id, in string uid)
 raises(NoSuchSupplier, NoSuchType);
 };

interface TypedProxy {
 void typedProxyRegister(
  in string type_id,
  in boolean relay,
  in NVList *arglist,
  in short argno,
```

```
  in string filter,
  in ConsumerSpec consumer,
  out string uid,
  out string eid)
 raises(RegistrationFailed);

 void typedProxyNotify(
  in string type_id,
  in short argno,
  in NVList *arglist)
 raises(NotConnected);

 TypedProxy typedProxyLookup(
  in string type_id)
 raises (NotFound);
 };
};
```

The semantics of a mediator (either generic or application-specific) depends on the types of suppliers, which can be simple or sophisticated. To support a simple supplier, a mediator will not register events at the suppliers. It accepts any event from the registered supplier, matches it against the registered event templates and notifies the consumers. To support a sophisticated supplier, a mediator can relay event registrations to the suppliers as required or process the events as for a simple supplier. To relay, a mediator does nothing but register the consumer's reference and assign a **user_id** to the consumer; the **user_id** is useful for the consumer to deregister its interest, and for the mediator to tell which consumer the received event belongs to. After this, the mediator invokes the **register<T>** method at the supplier with its own reference (instead of the consumer's reference) and the **user_id**, and then waits for notification. Upon notification, the mediator relays the notification to the consumer by invoking the **notify<T>** method at the consumer. Upon registration and notification, the mediator needs to construct a specific interface for registration e.g. **register<T>** at the supplier, and a specific interface for notification e.g. **notify<T>** at the consumer in case a **TypedProxy** is used. In both cases, the supplier should inform the mediator about the event types it notifies before the mediator accepts any registration from a consumer for the events.

3.3 Composite Event Service

There is an increasing demand for using composite events, for example, in telecommunication network management, several alarms raised by some network devices may contribute together towards a particular network problem known to the network manager; this requires that several events (i.e. alarms in this case) be signalled as a whole to the consumer (i.e. the network manager in this case). A composite event server is therefore included in our architecture as one of the main components. Specification of composite events needs to follow a well-defined syntax to allow standard parsing by a composite event server. A composite event algebra has been developed at Cambridge on which a composite event language is based. For instance, a sequence of events **A** and **B** is specified as **A** ; **B**, and event **A** or event **B** happens is specified as **A** | **B**.

One way to register a composite event with a server is using an application-specific interface. Typical parameters in such an interface include event type name, a list of parameters associated with the event in which each parameter is represented by a structure (e.g. NamedValue in CORBA) with attributes such as parameter name, parameter type, parameter value, parameter mode (i.e. in, inout or out), a filtering string, a string expression of the composite event (e.g. **A** | **B**) in the Cambridge Composite Event Language, and other parameters such as the consumer's reference, duration, QoS etc. The interface looks like:

registerComp < **CT** >(**string** $event_1$, **NVList** $parameters\_list_1$, **string** $filter_1$, ..., **string** $event_n$, **NVList** $parameters\_list_n$, **string** $filter_n$, **string** *expression*, **Duration** *d*, ...);

where **CT** is the type name of a composite event. As before, the position of the characters in the filter corresponds to the position of each of the parameters in the list.

Another possible way to register a composite event is using a standard interface in which expressions of composite event are specified in a well-defined syntax (e.g. the constraint language from the OMG Life Cycle Service [14]) and passed as a string. For example, a composite event may be expressed as:

"**event_type** = "enter"; **room** ="T14"; **person** = "Oliver"; **duration** = "Mon to Fri";" | "**event_type** = "absence"; **room** = "T14"; **person** = "*" ;"

The interface may look like:

registerComp(string *expression*, **Duration** *d*, ...);

We concentrate on the former because it is consistent with our interface design in COBEA. It is also useful to have a generic interface for composite event registration as it is for base events. To be notified of a composite event, a consumer has to submit upon registration the parameters to be passed through the notification. If the base events are not available at the server, the server will look for the suppliers; this is similar to the lookups by a consumer for a supplier. The interface is as follows.

```
module CompositeEventServer {
exception SyntaxError {};
interface CompAdmin:TypedMediator::TypedAdmin {};

typedef struct BaseEvent
 {string type_id; NVList *arglist; string filter};

typedef sequence<BaseEvent> CompEvent;

//generic composite event registration
 void typedRegisterCompEvent(
  in short eventno, //the number of base events
  in CompEvent comp_event, //related base events
  in string expression,//describes the comp. event
  in short out_argno,
  in NVList *out_arglist,
  in ConsumerSpec consumer,
  out long uid,
  out long eid)
 raises (SyntaxError,RegistrationFailed);

  void typedNotifyCompEvent:
   TypedMediator::TypedProxyNotify {};
  };
```

The CompAdmin is for a supplier to register itself with the server by indicating the base events it supports, and to get a reference to a proxy for passing the base events. There is no difference from a supplier's point of view whether a base event is used in a composite event or not.

Upon a registration of a composite event, the server will analyse the parameter **comp_event** to retain the type name, the parameters and the filters for each of the base events. The relation of the base events is obtained from the **expression**, e.g. **A;B;C** where **A, B** and **C** are base event type names. The server also retains from **out_argno** and **out_arglist** the parameters for constructing a notification interface to invoke at the consumer. More complex filtering is possible given the support for composite events. For example, consumers may specify a list of parameter values in events to be received ; a composite event **A**(12, "foo", "<===") | **A**(14, "foo", ">=== ") may be used for an event filter which checks if the first parameter is less than 12 or larger than 14, and the second parameter is "foo". Note that the composite event is expressed here intuitively rather than by using the interfaces defined in this section.

4 Building Applications with COBEA

In this section we list some application scenarios supported by COBEA.

4.1 Application Scenarios Supported

Scenario 1
An application creates a mediator object as a notification service with a generic interface for event registration and notification. OrbixTalk [10] and the TINA notification service can be constructed this way in COBEA. This scenario is not well supported by either the CORBA Event Service or the Cambridge Event Paradigm, in the latter a composite event server would be used for this purpose. Examples of such scenarios can be found in [10, 13, 21].

Scenario 2
An application defines its own typed interfaces for events which can be supported by the class library implemented for the primitives in COBEA. The applications supported by the Cambridge Event Paradigm can all be constructed this way in COBEA. This scenario is not well supported by the CORBA Event Service, and not supported at all by the TINA notification service in which an intermediate server is enforced. Examples of the scenario can be found in [3, 11].

Scenario 3
An application defines its own proxy for typed events without inheriting from the generic **TypedProxy** interface. The CORBA typed

event channel can support this scenario but no provision for registration of events is available. Furthermore, three steps must be carried out for connection to a CORBA event channel: (1) get an object reference for a factory which returns the reference to the proxies; (2) get an object reference for the supplier/consumer proxy; (3) connect to the proxy. It is much simpler to connect to the mediator than to the event channel.

4.2 An Example of Telecommunication Application

Our preliminary experience of using COBEA for building distributed active application systems includes developing an alarm correlation system for network management in telecommunications. This work [13] shares motivation and scenarios with alarm correlation research being done in Nortel Technology [20], but offers a solution to the problem that differs in key design elements, and offers it on the COBEA platform. Alarm correlation including alarm filtering can occur at several levels in the progress of an event from the raising object (usually a network device) through any intermediate critical real-time controlling software to the network manager. These levels (in the order of the lowest to the highest) are: hardware element; real-time control; system management. Our focus is on the system management level.

Alarms can indicate possible problems (i.e. raise hypotheses), can confirm existing hypotheses or can be accepted (without change of state) by existing hypotheses or existing (confirmed) problems. We use the Composite Event Language to express the complex relations between alarms and problems/hypotheses by treating alarms as base events and problems/hypotheses as composite events. We employ the evaluation engine and composite event language parser implemented in order to monitor and trigger these composite events. The system manager supplies the alarms (base events) to the composite event server, which monitors the composite events (problems/hypotheses) and notifies the problem/hypothesis browser in the alarm correlation system. After a problem has been diagnosed, appropriate actions can be taken at the system management level for network restoration. The work has shown that it is feasible to support active alarm correlation using COBEA and the Composite Event Language, and has suggested directions to improve the language [13].

Currently, we are building a trial system that features a graphical user interface for registering, deregistering and browsing composite events, and an event generator which can generate events of any specified type at a given interval. Performance will be measured to see if the system is suitable for online real time alarm correlation. Some real-time issues are discussed in [7]. Our experience shows that an event mediator can be used as a front end active database that incorporates the existing network management information base which supplies the network configuration data, and as a proxy for some of the dumb devices which signal alarms but do not understand CORBA.

5 COBEA Performance and Other Issues

COBEA is a general event architecture for building distributed active systems. Its main goal is to allow scalability by reducing the volume of notifications. The goal is achieved by implementing efficient filtering at event source. Our initial measurement against the prototype implementation has shown that filtering at event source is crucial for the system to scale as the volume of events increases. Some domain specific issues such as real-time issues described by [7] are not particularly addressed by COBEA. COBEA could be tuned for specific domains if required.

These tests were run on two Digital Alpha AXP 3000 workstations connected by a 155Mbit ATM network. One (for the event source and/or the event sink) is AXP 3000/900 with 275MHz CPU; and the other (mainly for the event sink) is AXP 3000/300 with 150MHz CPU; both have dual-issue processors running the OSF V3.2D-1 operating system. The event system and applications were built with g++ 2.7.2 with -O2 optimisation. The load was light on both machines most of the time during the testing, although the Alpha AXP 3000/900 normally had about 50 users and around 400 processes. We run many sinks on the Alpha AXP 3000/300 to avoid causing significant delay on the shared Lab machine (Alpha AXP 3000/900).

Firstly, we conducted the latency tests to determine

# events registered	filter & copy (μs) / # consumers		
	1	10	50
10	4.9	28	140
50	5.9	32	N/A
100	6.7	33	N/A

Table 1: Event Filtering Latency for One or More Consumers

the latency of filtering at event source. Secondly we conducted volume tests to determine the impact of event volume increase upon the event server.

Table 1 shows the result of the latency tests. When events occur, the server tries to match them against the registered event templates. The latency of event template matching increased logarithmically as shown in columns 1 and 2. For instance, column 1 shows that as the number of registered events increased from 10 to 100, the latency was 4.9μs, 5.9μs, and 6.7μs for 10, 50, 100 registered events respectively, for an average of 1000 runs. The latency for one server and multiple consumers increased linearly as the number of consumers increased (as shown in row 1); this is due to preparing events to send to each consumer after template matching. However, as shown in Figure 5, the overall cost of filtering is small; only the matched events are copied. Event dispatching delay in the case of one source and one consumer was 39μs on average when 100 events were registered. The overall delay for event creation, template matching and event dispatching (i.e. moving the events to the "sendqueue") were 271μs. The total latency between the occurrence of an event and delivery to the consumer is estimated to be less than 1 ms.

The advantage of filtering at event source is clearly shown in Table 2. We tested the effect of increasing event volume in two modes: *raw*, when no attempt to recover missing events was taken; and *normal*, when event sequence numbers were checked and attempts to recover missing events were made. When 10 events were registered by the consumer at the event source, the consumer detected no missing event at an event volume less than or equal to 2000Hz. In contrast, when 100 events were registered, events started to be missed when the volume reached 20Hz. If events were not recovered after loss, all registered events were correctly delivered to the consumer at the event rate as high as 8000Hz or beyond. Our test events were randomly generated integers between 1 to 1000. Parameterised filtering means that the consumer can register, say, number

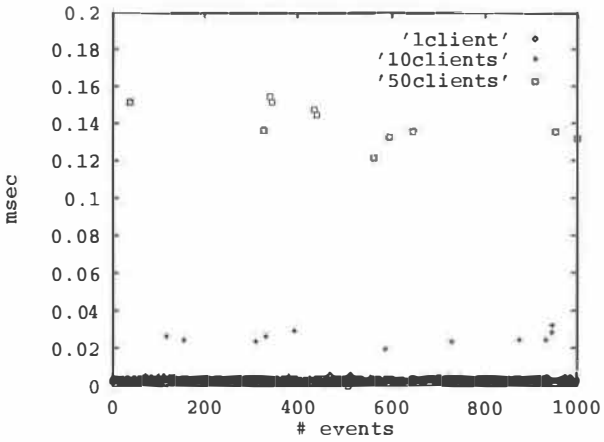


Figure 5: The Overall Cost of Filtering and Preparing to Send Events at Source

# events registered	100% delivery	
	raw (Hz)	normal (Hz)
10	> 8000	2000
100	20	10

Table 2: Event Volume from a Single Source to One Consumer

1 to 10 or as required. This is a real advantage over type-based filtering, in which case the consumer has no choice but to be overwhelmed by events.

It is interesting to notice that the cost of the integrated fault tolerance (i.e. sequence number checking and event recovering) was not as costly as we first thought, especially when the event volume and the number of registered events were low: i.e. at a volume less than 2000Hz when 10 events were registered. However, as the number of registered events increased, the system became much more sensitive to event volumes. Our experiments show that as 100 events were registered, about 98% of events were received by the sink in *raw* mode, while only 55% were received in *normal* mode. As the event volume goes extremely high, such cost will become expensive as it contributes to the load on both the source and the sink. In the case that many sinks try to recover missing events from a single event source, the source may eventually not be able to cope. COBEA provides solution to this by allowing replication of event sources and partition of the sinks, so each source is responsible to only a small number of sinks.

The current implementation may be improved for better scalability by allowing multicast of events,

thus removing the need to copy events. One potential obstacle to scalability is the integration of fault tolerance irrespective of the underlying communication platform; the cost can be eliminated if the communication support is reliable.

5.1 Other Issues

- **Dynamic Addition of New Event Types or Data**
  Dynamic addition of new event types is supported by using the generic typed interfaces defined in COBEA. Dynamic attachment of application-specific data to a generic event is supported by using the parameter **event_body**. Any type of data may be cast into this CORBA **any** parameter, which has two fields: **_type** and **_value**; **_type** can be checked in order to get the **_value** correctly.

- **Working with the CORBA Event Service**
  COBEA may be implemented alongside, or as an extension, to the CORBA Event Service. We have chosen the former for our current approach. A better alternative might have been to extend the CORBA Event Service interfaces to incorporate the new features supported in COBEA for standardization purposes. For example, the COBEA **snk** interface may be defined as follows:

```
#include CosEventComm.idl
module BaseEvent{
      ...
interface Snk:CosEventComm::PushConsumer{
 void notify(
  in EventHeader e,
  in any data)
 raises (NotConnected);
 void disconnec_snk();
 };

interface Src { ... };
};
```

  Also as mentioned earlier on in this paper, the pull model can easily be supported in COBEA by simply incorporating the pull interfaces of the CORBA Event Service. For example:

```
#include CosEventComm.idl
module BaseEvent{
      ...
interface COBEAPullSupplier:
 CosEventComm::PullSupplier{};
      ...
};
```

- **Event Naming and Locating**
  In COBEA, event type names are the same as interface type names, therefore a trader can be used to handle event naming and location; an event with parameters is like a service with attributes. If consumers are concerned with the content of an event (i.e. particular values of parameters), e.g. IBM stocks and Microsoft stocks of the **StockQuote** event, filters must be used to receive the quote of a particular stock only; a trader is not enough here.

- **Event Buffering and Logging**
  It is possible for events to happen before interest has been registered. Sometimes, a consumer can not digest all the events being supplied. It is therefore useful to buffer events at suppliers or mediators. A Time-To-Live (TTL) parameter can be associated with an event instance to make sure an event will not be discarded too quickly by the supplier or the mediator. It is useful to allow the consumers to specify a TTL. For some applications, events should be logged or made persistent if they may be subject to frequent query later. For instance, a security audit server may want to monitor logins by users, and log those events as evidence in case somebody attempts to use unauthorised resources.

- **Service Configuration**
  We propose a hierarchical structure (i.e. a rooted acyclic graph) for organising the servers which provide an event service cooperatively. Events generation may be partitioned among the servers, thus if a server does not know about a certain event itself, it can get help from the server which knows. In Figure 6, five servers are responsible for supplying events partitioned in the event groups Gr, G1, G2, G3 and G4 respectively.

- **QoS**
  In COBEA, a priority parameter in the event header can be used to specify the priority of an event. In addition, event queues are maintained according to priority for each event type, and events are sent FIFO within a priority. The **QoS** parameter in the **register()** operation can be used to specify speed or mode of event delivery, i.e. fast or reliable mode; the former has no guarantee of delivery of the

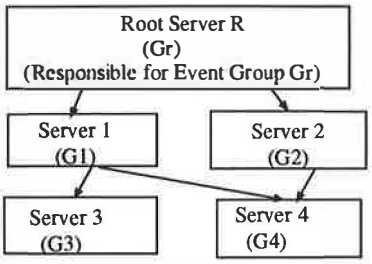


Figure 6: Composite event servers involved in handling a composite event

event, while the latter is implemented on top of a reliable transport protocol, thus can guarantee reliable delivery (e.g. events are delivered at least once and in order). Events are notified in the order of occurrence within a priority. The parameter may be used in many applications, for example, in telecommunication where fault events (or alarms) have to be delivered quickly so that the fault can be identified and rectified, however, performance events can tolerate some delivery delay as they are generally analysed off line at some later time.

- **Security**
  In many applications, events are required to be delivered or viewed only by authorised consumers. An object system where access control is based on object or method invocation only has not sufficient support for such requirements. Event-based access control requires that access control can be carried out against each event occurrence. On the one hand, suppliers or mediators (i.e. servers) are responsible for checking if an event can be delivered to the interested consumers. On the other hand, the servers must have the right to invoke the **notify()** operation at the consumers. In COBEA, a consumer must supply its user/role name using the **who** parameter when registering events. The parameter will be validated by the server to make sure that the consumer has the right to access the particular event. In order for the supplier to invoke the notification operation at the consumer, an ORB supporting secure method invocation, as specified by the recently adopted OMG CORBA Security Standard [16], can be used. Although many events have been excluded at registration time, more access control is still required in an event-based system. To avoid computation explosion in the order of **O**(no. of events × no. of consumers), coarse-grained access control based on object domain, event type, or method invocation only can be used as an alternative to the fine-grained control based on event occurrence. Failing this, optimised validation is still possible for access control depending on client credentials or event data only but not both.

6 Related Work

COBEA shares the major goals with the existing architectural frameworks for event handling in large distributed systems [2, 14, 18, 23, 24, 25]. The CORBA Event Service and the Cambridge Event Paradigm were reviewed above. Despite its weaknesses, the CORBA Event Service is nonetheless influential. Some recent work and products [7, 18, 10] have extended it; Expersoft, Iona, Sun Systems and Visigenic Software have developed commercial CORBA-compliant event services. The OMG TELECOM SIG has issued a Request For Proposal (RFP) on a notification service which has received several responses [18]. The proposed Notification Service must address issues such as filtering, assured notification delivery, security, QoS, and notification server federation. The Notification Service, however, is based on the indirect event communication model, and does not address implementation issues.

Work in real-time event notification [7, 19] has produced useful designs and implementations which use real-time threads for event publication in order to prevent priority inversion. Performance has been a major emphasis. [7] in particular, also address issues such as event filtering and correlation. However, its filtering and correlation mechanisms are not as powerful as the Cambridge Event Paradigm [2]. Moreover, filters can only be placed at the event channel.

Work in active databases is of direct relevance to the research on active systems [4, 5, 22]. In an active database, events include time events, start- and end-of-transaction events, operation invocation events, abstract events (events signalled from outside the database system). These are monitored and conditions are checked before actions are triggered upon event occurrences. The concept of composite events is introduced in active database for events correlation. Most techniques used for implementation are designed for a centralised database, thus do not address directly issues required for a distributed

implementation.

7 Concluding Remarks

We focused on the design of an architectural framework for event handling and showed how COBEA can be used to build large-scale distributed active systems under various application scenarios. We also reported our preliminary experience of using COBEA for building real applications.

COBEA supports the fundamental building blocks for developing active event-driven systems, namely the primitives, the mediator and the composite event service. The primitives form the foundation of the COBEA event architecture, in which the publish-register-notify mode is well supported for efficient asynchronous event communication. The mediator decouples the supplier from the consumer by accepting events from the suppliers, and passing events only to the interested consumers. Thus the supplier and the consumer do not have to know each other in order to communicate events. The composite event service, in particular, provides a powerful means of composing events via a number of operators and a convenient interface for the user to specify composite events. The distributed implementation of the service includes an evaluation engine for composite events, events timestamped at source, event streams and fault tolerance in the form of a heartbeat protocol.

The work focuses on the Notification Model and features well-defined interfaces for event registration, notification and filtering. Future work for COBEA includes incorporating security measures and implementing support for reliable event delivery.

Acknowledgements

This work is funded by EPSRC and Nortel Technology. We gratefully acknowledge the support from Nortel for developing the application scenarios in telecommunications, especially the help of Niall Ross. We would like to thank Richard Hayton, Christof Karl for implementing the components of COBEA.

References

[1] J. Bacon, J. Bates, and D. Halls. Location-oriented multimedia. *IEEE Personal Communications*, 4(5):48–57, October 1997.

[2] J. M. Bacon, J. Bates, R. J. Hayton, and K. Moody. Using events to build distributed applications. In *Proc. of the Second International Workshop on Services in Distributed Networked Environments*, pages 148–155. IEEE Computer Society Press, June 1995.

[3] John Bates and Jean Bacon. Supporting interactive presentation for distributed multimedia applications. *Multimedia Tools and Applications*, 1(1):47–78, March 1995.

[4] A. P. Buchmann et al. Building an integrated active OODBMS: requirements, architecture, and design decisions. In *Proc. of the 11th Intl. Conf. on Data Engineering*, Taipei, March 1995.

[5] S. Gatziu and K. R. Dittrich. Events in an active object-oriented database system. In *Proc. of the First Intl. Workshop on Rules in Database Systems (RIDS)*, Edinburgh, August/September 1993.

[6] W. Gaver et al. Europarc's RAVE System. In *Proc. of the ACM CHI'92 Conference on Human Factors in Computing Systems*, Monterey, Calif., 1992.

[7] T. H. Harrison, D. L. Levine, and D. C. Schmidt. The design and performance of a real-time CORBA event service. In *Proc. of the OOPSLA '97 conference*, pages 184–200, October 1997.

[8] Richard Hayton. *OASIS - An Open Architecture for Secure Interworking Services.* PhD thesis, Computer Laboratory, Cambridge University, June 1996.

[9] A. Hopper, A. Harter, and T. Blackie. The Active Badge System. In *Proc. of ACM INTECHI'93*, 1993.

[10] IONA. OrbixTalk - The White Paper. Technical report, IONA Technology, April 1996.

[11] JavaSoft. JavaBeans^{TM}. Version 1.00-A, December 1996.

[12] G. B. Kendon and N. F. Ross. Alarm correlator prototype: Demonstration script. Nortel Technology Internal Report, February 1996.

[13] Chaoying Ma. An Alarm Correlator Based on the Cambridge Event Technology. Active Systems Project (Cambridge University and Nortel Technology) Internal Report, April 1997.

[14] OMG. Common Object Services Specification, Volume I. OMG Document No. 94-1-1, March 1994.

[15] OMG. The Common Object Request Broker: Architecture and Specification. Revision 2.0, July 1995.

[16] OMG. CORBA Security. OMG Document No. 96-08-03 through 96-08-06, July 1996.

[17] OMG. Messaging Service: RFP. OMG Document No. ORBOS/96-03-16, March 1996.

[18] OMG. Notification Service: RFP. OMG Document No. Telecom/96-11-03, Nov. 1996.

[19] Ragunathan Rajkumar, Mike Gagliardi, and Lui Sha. The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation. In *First IEEE Real-Time Technology and Applications Sympo sium*, May 1995.

[20] N. Ross and K. Burn-Thornton. Design for Saffron Alarm Correlator. Nortel Technology Internal Report, February 1996.

[21] Douglas C. Schmidt and Steve Vinoski. The OMG Event Service. *C++ Report*, 9, Feb 1997.

[22] Scarlet Schwiderski. *Monitoring the behaviour of distributed systems.* PhD thesis, Cambridge University Computer Laboratory, April 1996.

[23] TIBCO. Tib/Rendezvous: White Paper. Technical report, TIBCO Software Inc., 1994.

[24] TINA-C. Engineering modelling concepts (DPE architecture). Technical Report TB_NS.005_2.0_94, TINA-C, December 1994.

[25] J. Warne. Event management for large-scale distributed systems. Technical Report APM.1633.01, APM Ltd., November 1995.

Objects and Concurrency in Triveni: A Telecommunication Case Study in Java

Christopher Colby*, Lalita Jategaonkar Jagadeesan†, Radha Jagadeesan*, Konstantin Läufer*, Carlos Puchol†

*Department of Math and CS, Loyola University Chicago
6525 N. Sheridan Road, Chicago, IL 60626
{colby,radha,laufer}@cs.luc.edu, http://www.cs.luc.edu/~{colby,radha,laufer}

†Bell Laboratories, Lucent Technologies
1000 E. Warrenville Road, Naperville, IL 60566
{lalita,cpg}@research.bell-labs.com, http://www.bell-labs.com/~{lalita,cpg}

Abstract

We describe the interaction of objects and concurrency in the design of Triveni, a framework for concurrent programming with threads and events. Triveni has been realized as JavaTriveni, a collection of tools for the Java programming language. We describe our experiences in JavaTriveni with an example from telecommunication.

1 Introduction

We describe the language-independent architecture of Triveni, a process-algebra-based design methodology that combines threads and events in the context of object-oriented programming. Triveni is compatible with existing threads standards such as Pthreads and Java threads, and with event models based on the Observer pattern. In particular, Triveni allows existing threads in the host language that conform to an Observer-pattern-based interface to be used as subcomponents. Dually, Triveni processes can be used as embedded systems in the host programming language if communication is arranged via the registration and notification mechanisms of the Observer pattern.

We have realized Triveni in Java as an API, JavaTriveni, that also includes an environment for specification-based testing; the detailed algorithms and design of JavaTriveni are described in [CJJ+98].

We present here the general design methodology underlying Triveni, using JavaTriveni as a concrete example. We also describe a case study in JavaTriveni, involving the re-implementation of a piece of telecommunication software, the *Carrier Group Alarms (CGA)* software of Lucent Technologies' 5ESS switching system.

Organization of the paper Section 2 describes the rationale and basis of Triveni. Section 3 gives a pattern-based description of the design methodology of Triveni; this discussion is illustrated concretely via the design of a game using Triveni. Section 4 describes our case study and includes a comparison with our earlier work [JPVO96] on this telecommunication software.

2 Triveni: Basis

Triveni is a programming methodology for concurrent programming with threads and events. Triveni has its basis in process algebras (e.g., CCS [Mil89], CSP [Hoa85]) and synchronous programming languages (e.g., see [Hal93]). The key feature of these formalisms is a notion of abstract *behavior*, which in a concurrent system is essentially the interaction of the system with its environment. Communication is via (labeled) events that are abstractions of names of communication channels. Triveni has the following features:

- Programs can be combined freely with the Triveni combinators, and one need only be concerned about the desired effects on the resulting behavior. Thus, Triveni combinators operate on behaviors and the result of the combinators are behaviors: the implementation of Triveni yields the correct combination of behaviors.
- Triveni enables parallel composition to be used freely for the modular decomposition of designs. In particular, the parallel composition of Triveni programs yields programs that are indistinguishable from simple ones (in much the same way that an object built by object composition has the same status as a simple object).

  The correct wiring among events sent by parallel components is done automatically by Triveni, and thus, the implementation of a program can closely reflect its design. Namely, each parallel component can be implemented separately: Triveni realizes the desired communication among them.
- Triveni supports exceptions via preemption combinators. For example, the watchdog combinator `DO P WATCHING e` yields a process that behaves like `P` until event `e` happens, upon which execution of `P` is terminated (in the spirit of "Ctrl–C"). Analogous to exception mechanisms in traditional programming languages, the preemption combinators aid in program modularity; for example, the watchdog above avoids the pollution of `P` with information about the event `e`.

  In Triveni, exceptions have first class status — *any* event can be an exception and can be used in the place of `e` in the watchdog. This allows exceptions to play an integral role in the programming of systems.

  Priorities on events are achieved by nesting of the preemption operators; for example, the event `e2` has higher priority than the event `e1` in the program fragment `DO (DO P WATCHING e1) WATCHING e2`. These priorities are not fixed by Triveni; they are determined by the program/design text.
- Triveni is compatible with the extensive existing work in both the design and implementation of programming languages and the analysis of concurrent systems. In particular, Triveni integrates the aforementioned ideas into the context of object oriented programming. Furthermore, Triveni is compatible with existing threads standards such as Pthreads and Java threads, and with event models structured on the Observer pattern [GHJV95]. Finally, Triveni includes a specification-based testing environment that automates testing of safety properties.

3 Triveni: Design and Implementation

In this section, we describe the architecture of Triveni. A game called Battle, whose rules are summarized in Figure 2, is used as a running example throughout this section. We discuss the design of Triveni at an abstract level using descriptions somewhat in the style of design patterns. Finally, we present a concrete design of Battle.

3.1 Processes as Objects

In Triveni, the class `Expr` captures the abstract notion of behavior. `Expr` enriches the structure of the encapsulated state in objects in two ways. (Figure 1 summarizes the following discussion.)

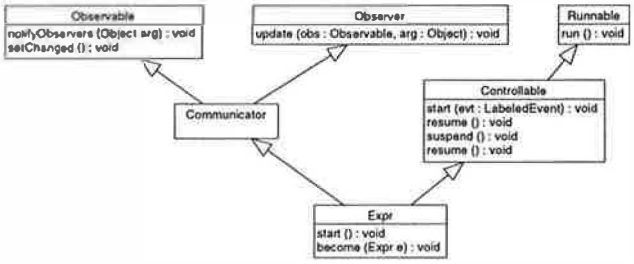


Figure 1: The Expr class

1. The `Communicator` interface captures reactivity, *i.e.* interaction with the environment. The environment uses the `Observer` interface to send inputs to `Expr` and the `Observable` interface to receive outputs from `Expr`. Thus, instances of `Expr` can be used as *embedded systems* in the host programming language if the communication is arranged via the Observer pattern.
2. `Expr` supports the encapsulation of autonomous state, such as system clocks, that can evolve even in the absence of interaction with the environment. The environment interacts with the

Battle is an n-player variation of the 2-player board game Battleship. New players cannot join the game once it has begun. A player loses by manually aborting the game or when all his/her ships are destroyed.

Oceans. Each player has a collection of ships on an individual ocean grid. The n ocean grids are disjoint. Each player's screen displays all n oceans, but a player can see only his/her own ships. A player's ships are confined to the player's ocean.

Ships. Each ship occupies a rectangular sub-grid of the player's ocean and sinks after each point in its grid area has been hit. There are two kinds of ships:

1. Battleships that can move on the surface of the player's ocean.
2. Submarines that can dive, but remain at a stationary position with respect to the player ocean's surface.

Moves. *A player can move as fast as the user-interface/reflexes allow.* Player i can make 4 kinds of moves:

1. Fire a round of ammunition on a square of another player j's ocean by clicking on it. The ammunition may hit a previously unhit point on one of player j's ships, in which case an **X** is displayed at that point in player j's ocean on all players' screens. No information is reported in case of a miss. The **X** marks are static; when a wounded battleship moves, or a wounded submarine dives, it does not affect previously displayed **X** marks on players' screens. When a ship is sunk, its position is revealed to all players.
2. Impart a velocity to a battleship that lasts until it receives another velocity command.
3. Make a submarine dive for a game-specific interval of time.
4. Raise a shield over his/her entire ocean for a game-specific interval of time, during which player i's ships are invulnerable. When a player raises an ocean-wide shield, his/her ocean becomes dim on the screens of all players. Each player has a limited supply of shields.

Figure 2: Rules of Battle

encapsulated autonomous program by the control operations indicated by the `Controllable` interface — started via `start()`, suspended via `suspend()`, resumed via `resume()`, and stopped via `stop()`. The `Controllable` interface corresponds closely to the control operations allowed on threads in Java — in particular, existing Java threads that conform to the `Communicator` interface for any event exchange can be used as `Exprs`.

The different kinds of state in `Expr` can interact. This discussion is best carried out in the context of a concrete example.

Example 1 *Consider the class of players in the Battle game. The user interface of the player is a reactive subcomponent of the player. The number of available shields can be modeled as an instance variable, say* `numshields`. *The timer that measures the duration of shielding evolves autonomously.*

The activation of the shielding (i.e. the initiation of the autonomous state) is caused reactively by inputs from the user interface. This activation affects the variable `numshields`. *The end of the period of shielding, as detected by the autonomously evolving clock object, causes a stimulus (in the form of brightening of this player's ocean) to the reactive subcomponents in other players.*

The `become` method in `Expr` follows standard object-oriented techniques. It allows an `Expr` to assume the behavior of another `Expr` and is useful for refining the inherited behavior in subclasses of `Expr`.

Example 2 *In the design of Battle that follows, a class called* `Ship` *is used to factor out the common behavior of Battleship and Submarine, namely the handling of opponent fire. (See Figure 3.) A* `Battleship` *is constructed from a* `Ship` *by adding instance variables and behavior to handle movement in terms of direction and speed. A sketch is as follows (detailed design is in Section 3.5):*

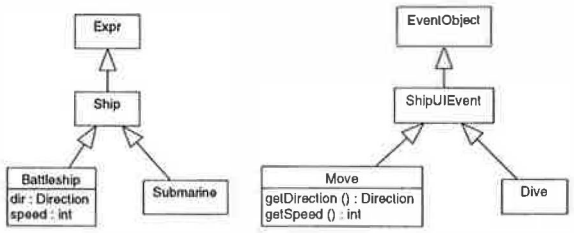


Figure 3: Inheritance Example

```
class Battleship extends Ship {
  Direction dir;
  int speed;
  Battleship(initial_status) {
    super(initial_status); // initialize receiver
    Expr e = // construction of new behavior
     // from inherited behavior
    become(e); // assume new behavior
}}
```

3.2 Building Triveni processes

The combinators that build Triveni programs are presented in Figure 4. The presentation as a *Composite* pattern leaves Triveni open to the addition of new combinators.

We first consider the `Activity` class. An `Activity` represents arbitrary code in the host programming language (say Java) that conforms to the interfaces `Communicator` and `Controllable`. The combinator `ActivityExpr` is used to embed an `Activity` in an `Expr` as its autonomous program. This allows the embedded `Activity` to be used as a subcomponent in the `Expr` and controlled by the `Expr`. In Triveni, the `Activity` class is actually a superclass (generalization) of the `Expr` class without the additional infrastructure that `Expr` provides for process composition.

```
public abstract class Activity extends Communicator
      implements Controllable { ... }
```

Example 3 *The GUI components for the user interface of the player in Battle, such as Player and Opponent windows, are best realized as* `Activities`. *This allows the GUI components to be embedded in the Triveni program for Battle as controllable subcomponents.*

The other combinators fall into the following categories. The Battle design example clarifies their semantics.

1. Triveni allows event-based communication — event emission (`Emit`), event renaming (`Rename`), and scoping in the form of local events (`Local`). Events are discussed in detail in Sections 3.3 and 3.4.

2. Triveni supports the classical constructions from process algebra — parallel composition (`Parallel`), sequential composition (`Sequence`), identity of sequential composition (`Done`), looping (`Loop`), waiting (potentially indefinitely) until a particular event happens (`Await`), and checking if the current event has a required label (`Present`).

3. Triveni also supports the preemption combinators from synchronous programming. This includes a watchdog (`DoWatching`) that terminates execution when a particular event happens, and a combinator that suspends the execution on a particular event and resumes it on another event (`SuspRes`).

4. In addition, Triveni provides structured interfaces (`Valuator`) to access the data carried on events and a combinator that branches on this information (`Switch`).

3.3 Events

In a Triveni program design, event labels are closely related to the class names in the event class hierarchy. This class-based view of labels induces an isomorphic hierarchy on the labels. This added structure makes renaming delicate; for example, the renaming of a label corresponding to a superclass has to propagate down the class hierarchy. However, it allows different parts of the system to view the same event object at different levels of granularity.

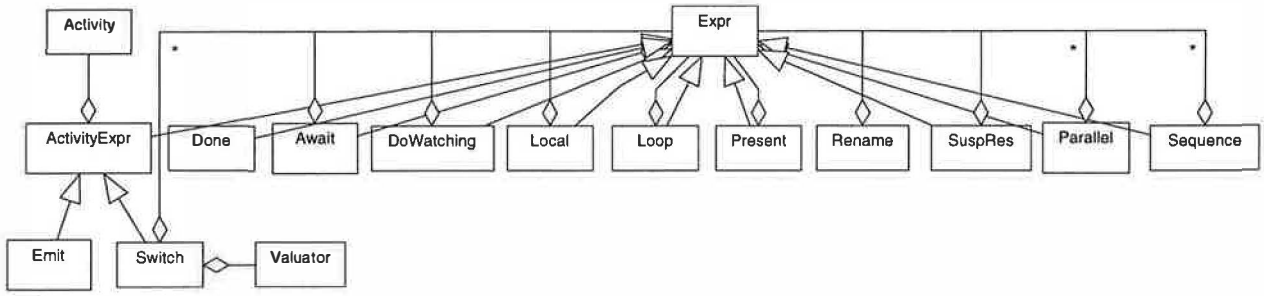


Figure 4: The Expr combinators as a Composite

Example 4 *In Battle, Figure 3 depicts the part of the event class hierarchy related to the class hierarchy for Ship, Submarine, and Battleship. The* `Battleship` *class handles* `Move` *events. The* `Submarine` *class handles* `Dive` *events. The presence of the class hierarchy on events allows the generalizing class, the* `Ship` *class, in our design to be set up in terms of the generalized event class, the* `ShipUIEvents` *class. This makes it independent of whether each one is a battleship, a submarine, or any other type of ship added later.*

Consider the following code fragment from Battle. There is a parallel composition (written `||`*) of several renamed instances of* `Ship` *along with the player's window* (`PlayerWindow`).

```
LOCAL ShipUIEvent_1, ..., ShipUIEvent_k IN
      PlayerWindow
   || RENAME [ShipUIEvent_1/ShipUIEvent] IN ship_1
      ...
   || RENAME [ShipUIEvent_k/ShipUIEvent] IN ship_k;
```

Thus, renaming on the event `ShipUIEvent` *in class* `PlayerOcean` *induces a renaming on the events* `Move` *in the* `BattleShip` *class and* `Dive` *in the* `Submarine` *class.*

3.4 Communication

In Triveni, the event delivery model is fair multicast: events are eventually and simultaneously delivered to all interested listeners. From an object point of view, one can view communication in Triveni as a refinement of the Observer pattern. Recall that in the Observer pattern, events are generated by event sources (subjects), and one or more listeners (observers) can register with a source to be notified about events of a particular kind. Triveni thus uses the registration and multicast mechanisms of the Observer pattern, but does not employ callbacks from the listeners back to the sources.

Triveni handles the registration of the Observer pattern by scoping mechanisms. In other words, every Triveni event has by default an associated scope established via the traditional programming language mechanisms such as local variables. A Triveni `Expr` then, by default, can listen to all events whose scopes include it.

Example 5 *Consider the code presented in example 4 above; this establishes* `k` *connections, one each between the* `PlayerWindow` *and each of the* `k` *ships.*

This "wiring" for event delivery is deduced from the program structure. The top level parallel composition sets up a group of Triveni processes that communicate via broadcast. The `local` *construct renders the outside world oblivious to the occurrence of the events of* `ShipUIEvent` *label (or variants thereof). Furthermore, in the concrete design later, ships are sensitive to only* `ShipUIEvents`*. Consequently, after renaming, the different ships occupy disjoint bands of the communication bandwidth leaving the* `PlayerWindow` *as the sole observer of each individual ship, and leaving each ship registered as an observer of only* `PlayerWindow`*.*

3.5 The Triveni program for Battle

Figure 5 shows a three-player Battle game, and Figure 6 shows the architecture of a Battle player.

The "wiring" in these figures represents the various kinds of events of the system: the tokens attached to the wires are event labels, and the event data fields, if any, are shown inside parentheses. A wire that

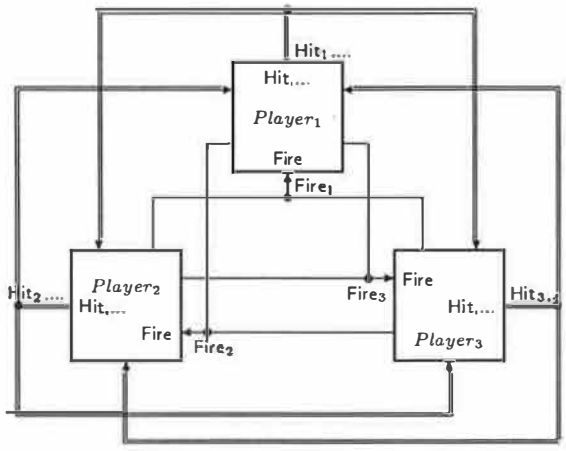


Figure 5: A three-player Battle game illustrating event-label renaming

has different names at each end represents an explicit event-label renaming. For example, Figure 5 shows that to connect several players in a game, player i's generic event labels (e.g., `Fire`, `Hit`, etc.) are renamed to their corresponding event labels subscripted by i.

In the entire code for the Battle game, there is no explicit wiring for events. Instead, all events are broadcast throughout a parallel composition, and the Triveni constructs of event-label matching and scoping via `LOCAL` and `RENAME` provide the necessary "wiring" of event delivery.

There are four kinds of GUI components for each player, shown as ovals in Figure 6.

- *Abort Button:* One per player. Emits the event `Abort`.
- *Shield Button:* One per player. Emits the event `Shield`.
- *Player Window:* One per player. For $1 \leq i \leq k$, where k is the number of ships per player, emits either event $\texttt{Move}_i\texttt{(direction, speed)}$ or event $\texttt{Dive}_i$, depending on whether ship i is a battleship or a submarine. Accepts events `Hit(position)`, `Sunk(status)`, and `Status(status)`.
- `Opponent Window`: $n-1$ per player, for an n-player game. Emits event `Fire(position)`. Accepts events `Hit(position)`, `Sunk(status)`, `Shield`, and `Unshield`.

The user-interface components are "generic" although they are not parameterized by a player index. Through event-label renaming, Triveni allows the differentiation and connection of multiple instances of a generic component. Indeed, user-interface components such as Player Window are most naturally implemented as subclasses of `Activity` embedded in and controlled by suitable subclasses of `ActivityExpr`:

```
class PlayerWindowUI extends Activity {
 //...create the user interface for the player window
}

class PlayerWindow extends ActivityExpr {
  PlayerWindow {
    super(PlayerWindowUI); // embed the user interface
                           // within this Expr
}}
```

Player i is implemented as a parallel composition of the top-level components shown in Figure 6. Its pseudo-code realization in Triveni is shown below. To aid readability, we use the Triveni combinators in infix form rather than the implicit prefix form of section 3.2; for example, we use `DO .. WATCHING ..` instead of `DoWatching(.., ..)`, `A || B` for `Parallel(A,B)`, etc.

```
class Player extends Expr {
 Player(i) {
  Expr e = RENAME[Fire_i/Fire, Hit_i/Hit, Sunk_i/Sunk,
            Shield_i/Shield, Unshield_i/Unshield,
            Abort_i/Abort] IN
              DO
                  AbortButton
               || Shield(number, duration)
               || SUSPEND Shield [PlayerOcean]
                  RESUME Unshield
               || OpponentOcean(1) || ...
               || OpponentOcean(n)    // except i
              WATCHING Abort;
  become(e);
}}
```

This code performs the renaming shown in Figure 5. The whole process is wrapped inside a `DO-WATCHING` construct, which preemptively terminates player i upon receipt of an `Abort` event. This is indicated in Figure 6 as a small boxed X at the scope of the entire player. Since the GUI components of the system are implemented as `Activities`, they are fully controlled by their surrounding `ActivityExprs`. Therefore, when a player presses the abort button, the single `DO-WATCHING` construct above terminates each component of his/her GUI. The `SUSPEND-RESUME` construct is used to ensure that when a player raises a shield, his/her own ocean is suspended until the shield runs out.

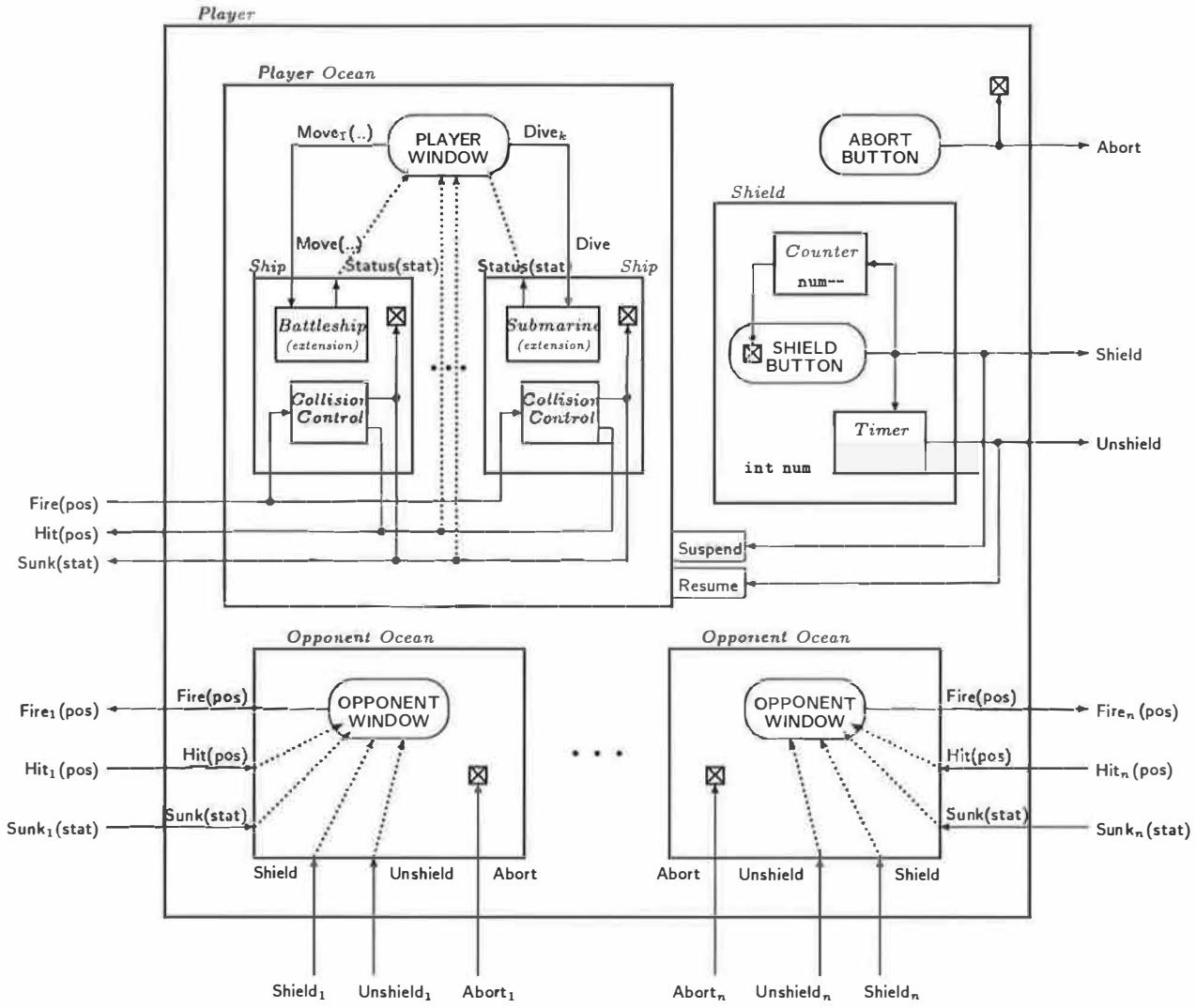


Figure 6: Architecture of a Battle player

While it is suspended, it will not respond to `Fire` events, but the player may still fire upon opponent oceans.

The player's shield process has an auxiliary timer `Activity` embedded in a subclass of `ActivityExpr`. This timer process will be reused throughout this example.

```
class Timer extends ActivityExpr {
  Timer(duration) {
    // accepts: Start
    // emits:   Finish
}}
```

The implementation of a timer is not shown; it is a generic timer that is tied to the shield button via the event-label renaming given below. The shield process comprises three components running in parallel:

1. The shield button, which is terminated upon receipt of an `OutOfShields` event.

2. A loop that decrements an instance variable `numshields` every time a shield is raised and emits an `OutOfShields` event when `numshields` reaches 0.

3. The shield timer.

The pseudo-code for the Shield process is as follows:

```
class Shield extends Expr {
 int numshields;
 Shield(number, duration) {
  numshields = number;
  Expr e = LOCAL OutOfShields IN
              DO ShieldButton WATCHING OutOfShields
           || LOOP Shield -> { numshields--; }
                             SWITCH (numshields == 0)
                             true: EMIT OutOfShields
                             false: DONE
           || LOOP
               RENAME [Shield/Start, Unshield/Finish]
               IN Timer(duration);
  become(e);
}}
```

The LOCAL hides the OutOfShields event from the rest of the system.

A player's ocean is parameterized by k ship processes, and is a parallel composition of all of them along with the player's window.

```
class PlayerOcean extends Expr {
 PlayerOcean(ship1, ..., shipk) {
  Expr e= LOCAL ShipUIEvent_1, ..., ShipUIEvent_k IN
             PlayerWindow
          || RENAME[ShipUIEvent_1/ShipUIEvent] IN ship1
             ...
          || RENAME[ShipUIEvent_k/ShipUIEvent] IN shipk;
  become(e);
}}
```

The code above is set up in terms of Ships and ShipUIEvents and exploits the inheritance hierarchy on Triveni objects and events, as illustrated in Figure 3. Thus, each one can be a battleship or a submarine.

Each ship is parameterized by a ShipStatus object that specifies its dimensions, position, and damage. When a ship process is started, it emits its status; these events are handled by the player window. Then (via the SEQ construct), it enters an event loop that reacts to Fire events, each carrying position data. The update method updates status upon a hit.

```
class Ship extends Expr {
 ShipStatus status;
 Ship(initial_status) {
  status = initial_status;
  Expr e = DO
           EMIT Status(status)
           SEQ
           LOOP Fire(pos) ->SWITCH(status.update(pos))
                           Hit: EMIT Hit(pos)
                           Sunk: EMIT Sunk(status)
                           Miss: DONE
         WATCHING Sunk;
  become(e);
}}
```

Battleships and submarines are implemented as subclasses of ship as illustrated in Figure 3, and share the above collision-control behavior.

A battleship contains two new instance variables, dir and speed, and adds a process in parallel with a generic ship process to handle Move events. At any point in time, a battleship is either stationary (speed is 0) or mobile. In the stationary state, it is awaiting an appropriate Move event to trigger a local Mobile event. In the mobile state, it invokes the move method of status at intervals of 1/speed until it becomes stationary again. Both the battleship and submarine processes reuse the timer process, originally introduced for the shield process above.

```
class Battleship extends Ship {
 Direction dir;
 int speed;
 Battleship(initial_status) {
  super(initial_status);
  Expr e = DO
              this    // behavior inherited from Ship
           || LOCAL Stationary,Mobile,Start,Finish IN
                    LOOP Move(d,s) ->
                           { dir = d; speed = s; }
                           SWITCH (speed == 0)
                              true: EMIT Stationary
                             false: EMIT Mobile
                || LOOP
                      DO
                        AWAIT Mobile ->
                           LOOP
                             EMIT Start
                             SEQ
                             AWAIT Finish ->
                                {status.move(dir)}
                                EMIT Status(status)
                           ||
                            LOOP Timer(1/speed)
                        WATCHING Stationary
         WATCHING Sunk;
  become(e);
}}
```

A submarine is a ship that is suspended upon receipt of a Dive event and resumed after some duration of time. Suspending a ship suspends the collision-control process and thus renders it invulnerable to attack.

```
class Submarine extends Ship {
 Submarine(initial_status, dive_duration) {
  super(initial_status);
  Expr e = DO LOCAL Start, Finish IN
                 SUSPEND Start [this]
                 RESUME Finish
              || LOOP Dive -> ( EMIT Start
                                SEQ
                                AWAIT Finish )
                 || LOOP Timer(dive_duration)
           WATCHING Sunk;
   become(e);
}}
```

An opponent ocean is an opponent window, with the events appropriately renamed to tie together with the opponent's process in a multiplayer game. If an opponent aborts the game, this will cause his/her corresponding ocean on the screens of all other players to disappear. Since `Shield` and `Unshield` events are broadcast each player knows when an opponent has raised a shield.

```
class OpponentOcean extends Expr {
 OpponentOcean(j) {
  Expr e = RENAME[Fire_j/Fire, Hit_j/Hit, Sunk_j/Sunk,
                  Shield_j/Shield, Unshield_j/Unshield,
                  Abort_j/Abort] IN
              DO OpponentWindow WATCHING Abort;
  become(e);
}}
```

An n-player game is simply constructed by composing n player processes in parallel.

3.6 The Implementation of JavaTriveni

We have implemented Triveni in Java as a class library. The design of the JavaTriveni implementation and the underlying algorithms are described in [CJJ+98]. The relationship between class names and event labels in Triveni must currently be established by the application programmer and is not currently enforced by the system.

Here, we briefly sketch the architecture of a Triveni process, referring the reader to [CJJ+98] for details. The implementation of a JavaTriveni process P comprises of a controller C_P, which is a deterministic state machine, and a multiset of concurrent communicating activities ($\{A_{P,1}, \ldots, A_{P,n}\}$), possibly implemented in the host language Java. In particular, event emissions are realized as activities. Every transition in the state machine C_P is labeled with an event name and a set of side-effects that will occur when this transition is taken – these side effects can include control operations on activities via the `Controllable` interface, such as `start()`, `suspend()`, `resume()`, and `stop()`. A given transition labeled e is triggered upon receipt of an event with label e if the current state of the state machine is the source state of the transition. C_P also controls all communication between its activities — each activity $A_{P,i}$ emits events to C_P, which may forward it back to one or more selected activities $A_{P,j}$. The implementations of all Triveni combinators operate on such structures and yield such structures.

Our JavaTriveni implementation includes a non-intrusive form of instrumentation for testing and debugging in the flavor of `assert` statements in traditional languages. In particular, system specifications can be expressed as safety properties; informally, these properties stipulate that "something bad never happens." Temporal logic is a well-known formalism for specifying safety properties, and our specification language is based on its propositional linear-time variant [MP92]. This notation provides a straightforward means of expressing conditions on sequences of events.

Our implementation uses the following fact about safety properties: for any safety property, there exists a finite-state machine whose language is the set of all possible (finite) executions that violate the property. From the given property, our implementation automatically generates a JavaTriveni process, which encodes this finite-state machine. This process is composed in parallel with the process that is being monitored. If the specified property is violated at any point during an execution of the system, the above JavaTriveni process generates a special event, and the assertion fails. The user has the option to abort the application, ignore the failed assertion, or ask the system to report entire test traces.

4 A Telephone Switching System Application

We now describe our telecommunication case study in JavaTriveni.

Lucent Technologies' 5ESS telephone switching system [MS85] is a concurrent reactive system comprised of millions of lines of C code. In this switch, a wide variety of carrier group types are used to transmit data corresponding to end-to-end telephone connections. These carrier groups are attached to various hardware units on a set of processors, which are responsible for routing telephone calls. Malfunctions on these carrier groups, such as lost framing, lost events, or physical accidents, can result in disturbance or abrupt termination of existing phone calls. The *Carrier Group Alarms (CGA)* software in the 5ESS switch is responsible for reporting status changes — malfunctions or recoveries from malfunctions — on carrier groups, so that other 5ESS software can respectively remove or restore the as-

sociated carrier groups from service, and route new telephone calls accordingly [HLRW85].

As a case study, we have re-implemented part of the CGA software in JavaTriveni. The starting point of our implementation and the top level design come from our earlier work [JPVO96]. We repeat here our earlier description and design of the CGA software [JPVO96] in order to keep this paper self-contained. For proprietary reasons, the descriptions of our version given in this paper do not reflect the specific details of the actual 5ESS switch software, and we note that the JavaTriveni code in this paper is not part of the 5ESS switch.

One of the main sources of inputs to the CGA software are *summary requests* from either human operators or some other parts of the switch. In response, the CGA software must collect data about the status of all the carriers on all the relevant processors, and print this information on various consoles and printers via the *Human-Machine Interface (HMI).*

One component, called the "CGA Collection Software," requests every relevant processor to send data about the status of all the carrier groups attached to that processor. This software then formats the received data in a manner suitable for printing on various consoles and printers via the HMI. The other components, called the "CGA Data Software," reside on the processors on which the carrier groups are attached. When a request for data arrives from the CGA Collection Software to the CGA Data Software on a given processor, this processor searches the relevant databases for status information on all the carriers that are attached to that processor. The data is then sanity-checked — namely, that this particular sort of status change can actually occur on the given carriers and is not merely the outgrowth of a database error. The data is collected into a packet and sent to the CGA Collection Software, after which this instance of the CGA Data Software waits for the next request from the CGA Collection Software. After receiving the next request, it resumes searching for more data, from the point it left off in the corresponding databases. When all the relevant data has been gathered, an appropriate termination message is sent to the CGA Collection Software. All communication between the CGA Collection Software and the instances of the CGA Data Software is through asynchronous message passing.

There are a number of issues that we needed to consider in writing our JavaTriveni version (and our earlier version) of the CGA software. For example:

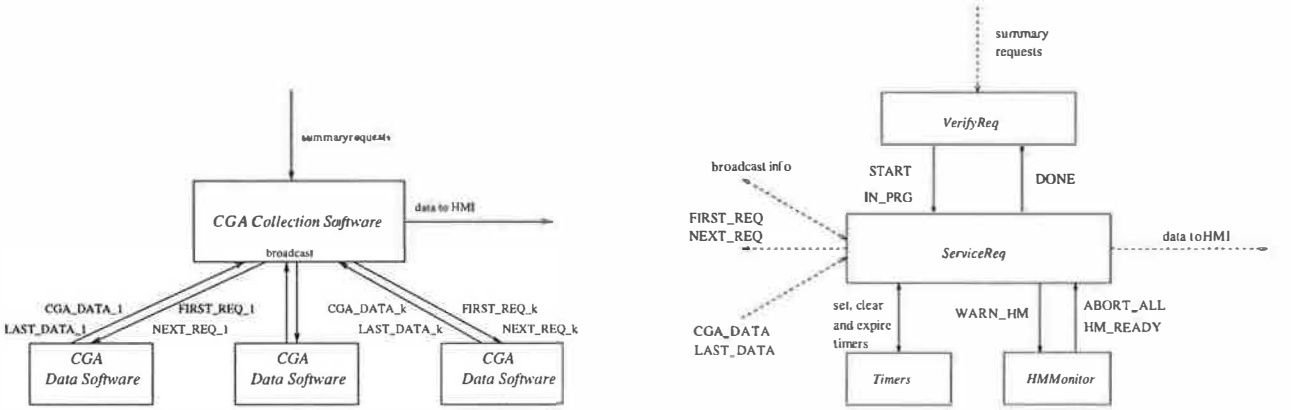
- What should be done if a processor does not respond to a request for data?
- Should the CGA Collection Software keep sending requests for data to the processors if the HMI is not responding?
- Should more than one summary request ever be in process simultaneously?
- Is there a way to terminate a summary request prematurely?

We have dealt with these problems in quite a natural manner, thanks to the expressive power of JavaTriveni. Our case study version is described below. In this case study, we follow closely our earlier design [JPVO96].

4.1 JavaTriveni version of the Carrier Group Alarms software: Structure and Advantages

Our version of the CGA software consists of approximately 2500 lines of concurrent code in JavaTriveni. The functionality of this software, comprised of the CGA Collection Software and multiple instances of the CGA Data Software, is depicted in Figures 7 and 8. (The structure of the CGA Data Software is relatively simple, and hence is depicted merely as pseudo-code). Arrows emanating from Triveni processes indicate events that are emitted or flags that are set by those Triveni processes; arrows pointing to Triveni processes indicate events that are received or flags that are read by those Triveni processes. Dotted arrows represent events to or from the outside world.

The CGA Collection Software receives summary requests from the outside world. In response, it first broadcasts a message to all the instances of the CGA Data Software to start collecting their data. It then sends requests to the multiple instances of the CGA Data Software; these instances are polled sequentially. The first request from the CGA Collection Software to a given instance of the CGA Data Software is represented by the `FIRST_REQ_i` events, and subsequent requests are represented by `NEXT_REQ_i`




```
class CGA extends Expr {
 CGA(CollectionSoft, DataSoft1, ..., DataSoftk) {
  Expr e=     CollectionSoft
           || RENAME[CGA_DATA_1/CGA_DATA,
                     LAST_DATA_1/LAST_DATA,
                     FIRST_REQ_1/FIRST_REQ,
                     NEXT_REQ_1/NEXT_REQ] IN DataSoft1
                ...
           || RENAME[CGA_DATA_k/CGA_DATA,
                     LAST_DATA_k/LAST_DATA,
                     FIRST_REQ_k/FIRST_REQ,
                     NEXT_REQ_k/NEXT_REQ] IN DataSoftk
  become(e);
}}
```

```
class CollectionSoft extends Expr {
 CollectionSoft() {
  Expr e=     VerifyReq
           || ServiceReq
           || Timers
           || HMMonitor
  become(e);
}}
```

Figure 7: Architecture of CGA Software(left), CGA Collection Software (right)

events. The given instance of the CGA Data Software responds to a FIRST_REQ_i event by collecting a threshold amount of data from the beginning of its databases, and sending CGA data to the CGA Collection Software via the CGA_DATA_i event. It then waits for a NEXT_REQ_i event, upon which it resumes searching for more data, from the point it left off in the corresponding databases. It again sends data via the CGA_DATA_i event. The LAST_DATA_i event signifies that all relevant CGA data has been sent by this instance of the CGA Data Software. The CGA Collection Software collects all the CGA data, reformats it, and sends it to the Human-Machine Interface for printing.

```
class DataSoft extends Expr {
 DataSoft() {
  Expr e= AWAIT FIRST_REQ ->
          LOOP
            // collect threshold amount of data
            // from beginning of database
            // emit CGA_DATA or LAST_DATA
            DO
              LOOP
                AWAIT NEXT_REQ ->
                  // collect threshold amount of data
                  // data from rest of database
                  // emit CGA_DATA or LAST_DATA
            WATCHING FIRST_REQ
  become(e);
}}
```

Figure 8: Design of the CGA Data Software

The JavaTriveni design and implementation of the top-level CGA program, the CGA Data Software, and the CGA Collection Software utilize the principles underlying JavaTriveni. In particular:

1. The CGA sub-programs are combined freely with the JavaTriveni combinators, and the JavaTriveni tools produce an implementation that yields the correct combination of behaviors. For example, all Triveni processes (denoted by boxes in the figures) are viewed as black boxes by the rest of the program, and the design and implementation of the CGA programs is based only on the desired effects on the resulting behavior.

2. Parallel composition is used freely for the modular decomposition of designs, and the JavaTriveni tools automatically implement the desired communication. For example, the modules in Figure 7 are composed using the Java-

Triveni `Parallel` construct and the desired wiring depicted in the figures is realized by JavaTriveni.

3. The preemption operators of JavaTriveni aid in program modularity and allow expressing priorities on events. For example, consider the CGA Data Collection software of Figure 8. It uses preemption to indicate that the `FIRST_REQ` event has higher priority than the `NEXT_REQ` event. Namely, the `AWAIT NEXT_REQ` statement occurs inside the `DO ... WATCHING FIRST_REQ` statement. This corresponds to the desired CGA functionality that if a `FIRST_REQ` event arrives — perhaps as a result of the previous request being aborted and a new request being started — then the database will be searched from the beginning for possible alarm data on this processor. The use of the preemption operators to express priorities avoids the pollution of the code following the `AWAIT NEXT_REQ ->` statement with information regarding `FIRST_REQ`.

4. The combination of objects, renaming, and inheritance gives a convenient way to express variances in program components in the places they are used. For example, in Figure 7, renaming of the events passed between the CGA Collection Software and the multiple instances of the CGA Data Software allow different communication channels to be used for the different instances.

The JavaTriveni design methodology is also evident at a "micro" level in the the following detailed description of the JavaTriveni implementation of the CGA Collection Software.

The architecture of the CGA Collection Software

The CGA Collection Software (Figure 7) has four parallel Triveni processes: *VerifyReq*, *ServiceReq*, *HMMonitor*, and *Timers*. Figures 9–11 show the internal structure of some of these Triveni processes. As before, the modules in the figures are composed using the JavaTriveni `Parallel` construct, and the desired wiring depicted in the figures is realized by JavaTriveni.

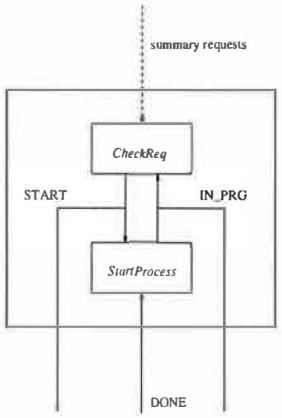


Figure 9: Internal Structure of *VerifyReq*

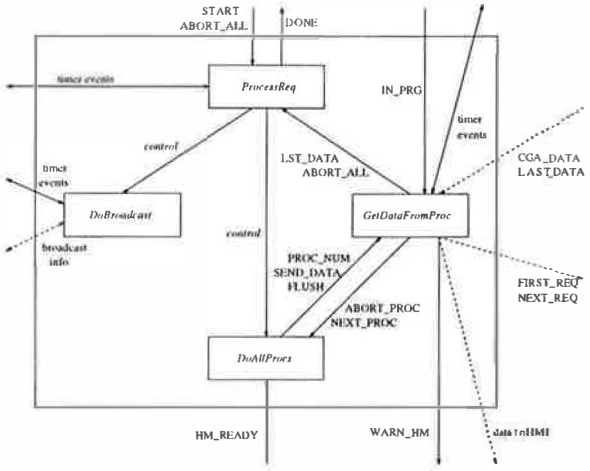


Figure 10: Internal Structure of *ServiceReq*

Verifying a Request Summary requests are first verified by the *VerifyReq* Triveni process, whose internal structure is illustrated in Figure 9. There are various types of *summary requests*, and each one has an associated internal `IN_PRG` flag that denotes that this particular type of request is currently in progress. The *CheckReq* Triveni process waits for summary requests, using the JavaTriveni `Await` construct. If some other request is in progress, i.e., the corresponding `IN_PRG` flag has been set by *StartProcess*, then the requesting party is asked to "retry later." Otherwise, the request is started, i.e., the `START` event is emitted by *CheckReq* and the appropriate `IN_PRG` flag is set by *StartProcess*.

Servicing a Request The `START` event and `IN_PRG` flag are received/read by the *ServiceReq* Triveni process, which is responsible for servicing the request. The internal structure of *ServiceReq*

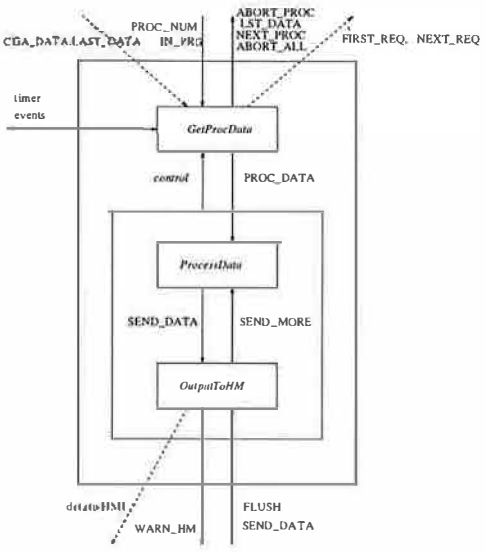


Figure 11: Internal Structure of *GetDatafromProc*

is depicted in Figure 10. The *ProcessReq* Triveni process waits for the START event and first sets a timer for the maximum amount of time that may be spent servicing a single request. This timer is set using the event TOTAL_TIMER; the events TOTAL_TIMER_EXPIRED and TOTAL_TIMER_CLEAR, respectively, indicate the expiration or clearing of this timer. The TOTAL_TIMER_EXPIRED event is of high-priority: in particular, upon receipt of this event, the current request, if active, is aborted, the event DONE is sent to *VerifyReq* and its internal Triveni processes, *StartProcess* resets the IN_PRG flag, and *CheckReq* starts accepting new requests. This form of process abortion is expressed using the JavaTriveni DoWatching construct in the *ProcessReq* module. This gives high-priority to the TOTAL_TIMER_EXPIRED event, while allowing the rest of the Collection Software to remain unpolluted by information about this event.

Alerting the Processors After the timer is set by *ProcessReq*, the *DoBroadcast* Triveni process becomes active and, in turn, sets another timer and broadcasts a command to all the processors to start collecting data. When the broadcast completes or this timer expires, the *DoBroadcast* Triveni process becomes inactive and the *DoAllProcs* Triveni process becomes active. This timer event is of lower priority than the TOTAL_TIMER_EXPIRED event. This is expressed through appropriate nesting of preemption operators: in particular, the Await construct for this timer event is nested inside the DoWatching construct for the TOTAL_TIMER_EXPIRED event.

Collecting Data from the Processors If the HM_READY flag is set, *DoAllProcs* gets the identifier of the first processor to be queried for data about carrier groups, and passes this identifier to the *GetDataFromProc* Triveni process as a value on the event PROC_NUM. Figure 11 illustrates the internal structure of the *GetDataFromProc* Triveni process. The PROC_NUM event is received by its internal Triveni process *GetProcData*, which then sets a timer and sends a FIRST_REQ event (or a NEXT_REQ event) to the corresponding processor, requesting data. If the timer expires before the processor replies (with a CGA_DATA or a LAST_DATA event), the query of this processor is aborted, ABORT_PROC is emitted, and *DoAllProcs* starts processing the next processor. (As before, this timer event is of lower priority than the TOTAL_TIMER_EXPIRED event, expressed through appropriate nesting of preemption operators.) Otherwise, when the processor replies, the data on the received event is sent to *ProcessData* as a value on the event PROC_DATA. *ProcessData* formats the data in a manner suitable for sending to the Human-Machine Interface. Pieces of data are sent individually to *OutputToHM* via SEND_DATA every time SEND_MORE is received. *OutputToHM* then sends the data to the HMI; if there is a resource overflow and a message is lost, *OutputToHM* sends a WARN_HM event to *HMMonitor*.

This cycle continues, using the JavaTriveni Loop construct, until the last piece of data is collected from a given processor (indicated by the LAST_DATA event), after which *GetProcData* emits the NEXT_PROC event. *DoAllProcs* then gets the identifier of the next processor to be queried, and the cycle is repeated until all the data on all the processors is collected. If there is a fatal error in packaging the data or accessing the HMI, the request is aborted by sending ABORT_ALL to *ProcessReq*. This event is of high-priority, and the resulting behavior is similar to that of the TOTAL_TIMER_EXPIRED event. In particular, control then returns to *ProcessReq*, the request is aborted, the event DONE is sent to *VerifyReq*, and its internal Triveni process *CheckReq* starts accepting new requests. The response to a problem in determining the first or next processor is similar, except that the ABORT_ALL event is not emitted.

When the last processor has been queried, *DoAllProcs* emits SEND_DATA and FLUSH to *OutputToHM*

so that any remaining data is sent to the HMI. *ProcessReq* then emits the event `DONE` so that *CheckReq* can start accepting new requests.

The Human-Machine Monitor Whenever a `WARN_HM` is emitted by *OutputToHM*, *HMMonitor* resets the `HM_READY` flag, and data collection from new processors is suspended by *ServiceReq*'s internal Triveni process *DoAllProcs*. This behavior is expressed using the JavaTriveni `SuspRes` construct inside the *DoAllProcs* module: this allows the rest of the Collection Software program to remain unpolluted by information about `HM_READY`, while giving high-priority to this event. *HMMonitor* then periodically checks if the HMI is responding. Once the HMI starts responding, data collection is resumed. If it does not respond in a threshold number of queries, *HMMonitor* sends an `ABORT_ALL` to *ServiceReq*'s internal Triveni process *ProcessReq*, and the summary request is aborted. The nesting structure of the `SuspRes` construct for `HM_READY` and the `DoWatching` construct for `ABORT_ALL` give the desired dynamic priorities among these events, depending on the number of times the HMI has been queried.

Timers Timers are set and cleared through the *Timers* Triveni process, which also sends events to the other Triveni processes when a timer has expired.

4.2 Testing of safety properties

In our earlier work [JPVO95], a 5ESS developer had provided a summary of safety properties that this variation of the CGA software should satisfy. We consider some of the same safety properties here.

The actual timing constants have been omitted here due to proprietary considerations and have been denoted by symbols c_i. These are so-called "soft" real-time properties in the sense that the exact bounds c_i need not be satisfied; a reasonable approximation will do.

- T0 A summary request must be completed in less than time c_1.
- T1 If a queried processor does not reply within time c_2, the request should be aborted immediately and the next processor should be queried.
- T2 If the HMI blocks on a message, the collection of new CGA data must suspend.
- T3 If the HMI blocks on a message, the message should be resent with a period of time c_3, until the HMI unblocks. If time c_4 elapses and the HMI has not yet unblocked, the summary request should be aborted.
- T4 If HMI unblocks after CGA data collection has been suspended, CGA data collection must be reactivated immediately.
- T5 No summary request should be honored when another summary request is currently running.

Using the specification-based testing facility of JavaTriveni, we have tested our JavaTriveni implementation of the CGA software against these properties. Since our JavaTriveni version used system timers to enforce timing constraints, our implementation can only be expected to satisfy the above properties under certain obvious assumptions about these system timers. In particular, we need to assume that when a timer is set with the value c_i, it either expires or is cleared within time c_i after it is set.

4.3 Comparison with earlier work

Our earlier work involved writing an implementation of the Carrier Group Alarms software in the synchronous programming language ESTEREL [BG92]. Both the ESTEREL and JavaTriveni versions of the program are about 2500 lines of code.

ESTEREL elegantly models simultaneous events, and in this regard is superior to JavaTriveni's simulation of simultaneity. In our JavaTriveni code, we followed the ESTEREL design closely; however, most assumptions on event simultaneity could safely be eliminated, and data flags were used to simulate simultaneity in the few remaining cases.

ESTEREL only supports very rudimentary notions of autonomous behavior and asynchronous communication. Thus, in our earlier work the *Timers* Triveni process of the JavaTriveni implementation was realized *outside* the ESTEREL framework, via an operating system call, and the the communication of the CGA Collection Software and the CGA Data Software was implemented using C system calls. In contrast, JavaTriveni fully integrates autonomous and

reactive behavior and supports asynchronous communication, and the entire summary request functionality of the CGA software was implemented in JavaTriveni.

5 JavaTriveni Distribution and Future Work

Information regarding the JavaTriveni distribution can obtained by contacting the authors.

A next step in Triveni is to study the interaction between the event-based exceptions and priorities in Triveni with Java's existing notions of exceptions and thread priorities.

The next phase of the Triveni project is the investigation of the interaction between Triveni and distributed programming, such as via remote method invocation (RMI) in Java.

Acknowledgments

We are grateful to Douglas Lea for several interesting and very useful discussions about this work, and for many useful comments on this paper. We thank James Coplien for many helpful comments on this paper. We also acknowledge the useful comments of anonymous members of the COOTS program committee.

Christopher Colby and Radha Jagadeesan were supported in part by CAREER awards from the National Science Foundation.

References

[BG92] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.

[CJJ+98] C. Colby, L. J. Jagadeesan, R. Jagadeesan, K. Läufer, and C. Puchol. Design and implementation of Triveni: A process-algebraic API for threads + events. In *Proceedings of the 1998 IEEE International Conference on Computer Languages*. IEEE Computer Press, 1998. To appear.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Hal93] N. Halbwachs. *Synchronous programming of reactive systems*. The Kluwer international series in Engineering and Computer Science. Kluwer Academic publishers, 1993.

[HLRW85] G. Haugk, F.M. Lax, R.D. Royer, and J.R. Williams. The 5ESS(TM) switching system: Maintenance capabilities. *AT&T Technical Journal*, 64(6 part 2):1385–1416, July-August 1985.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[JPVO95] L. Jagadeesan, C. Puchol, and J. E. Von Olnhausen. Safety property verification of ESTEREL programs and applications to telecommunications software. In *Lecture Notes in Computer Science*, volume 939, pages 127–140, July 1995. Proceedings of the 7th International Conference on Computer Aided Verification.

[JPVO96] L. Jagadeesan, C. Puchol, and J. E. Von Olnhausen. A formal approach to reactive systems software: A telecommunications application in ESTEREL. *Formal Methods in System Design*, 8(2):123–152, March 1996.

[Mil89] R. Milner. *Communication and Concurrency*. Series in Computer Science. Prentice Hall, 1989.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[MS85] K.E. Martersteck and A.E. Spencer. Introduction to the 5ESS(TM) switching system. *AT&T Technical Journal*, 64(6 part 2):1305–1314, July-August 1985.

An Object-Oriented Framework for Distributed Computational Simulation of Aerospace Propulsion Systems

John A. Reed and Abdollah A. Afjeh

The University of Toledo
Toledo, Ohio

{jreed, aafjeh}@eng.utoledo.edu

Abstract

Designing and developing new aerospace propulsion systems is time-consuming and expensive. Computational simulation is a promising means for alleviating this cost, but requires a flexible software simulation system capable of integrating advanced multidisciplinary and multifidelity analysis methods, dynamically constructing arbitrary simulation models, and distributing computationally complex tasks. To address these issues, we have developed Onyx, a Java-based object-oriented application framework for aerospace propulsion system simulation. The Onyx framework defines a common component object model which provides a consistent component interface for the construction of hierarchal object models. Because Onyx is a framework, component analysis models may be changed dynamically to adapt simulation behavior as required. A customizable visual interface provides high-level symbolic control of propulsion system construction and execution. For computationally-intensive analysis, components may be distributed across heterogeneous computing architectures and operating systems. This paper describes the design concepts and object-oriented architecture of Onyx. As a representative simulation, a set of lumped-parameter gas turbine engine components are developed and used to simulate a turbojet engine.

1 Introduction

As the aerospace propulsion industry moves into the 21st century, there is increasing pressure to reduce the time, cost and risk of jet engine development. To meet the harsh realities of today's marketplace, innovative approaches to reducing propulsion system design cycle times are needed. An opportunity exists to reduce design and development costs by replacing some of the large-scale testing currently required for product development with computational simulations. Increased use of computational simulations promise not only to reduce the need for testing, but also to enable the rapid and relatively inexpensive evaluation of alternative designs earlier in the design process.

As a result of these forces, several government-industry cooperative research efforts have been established to develop technologies that enable the cost-effective simulation of a complete air-breathing gas turbine engine. In the United States, the Numerical Propulsion System Simulation (NPSS) project has been established between the aerospace industry, Department of Defense, and NASA. When completed, NPSS will be capable of analyzing the operation of an engine in sufficient detail to resolve the effects of multidisciplinary processes and component interactions currently only observable in large-scale tests [1, 2]. For example, more accurate predictions of engine thrust and efficiency would be possible if the "operational" geometry of a compressor rotor, stator, and casing could determined based on an analysis of the combined aerodynamic, structural and thermal loadings [3].

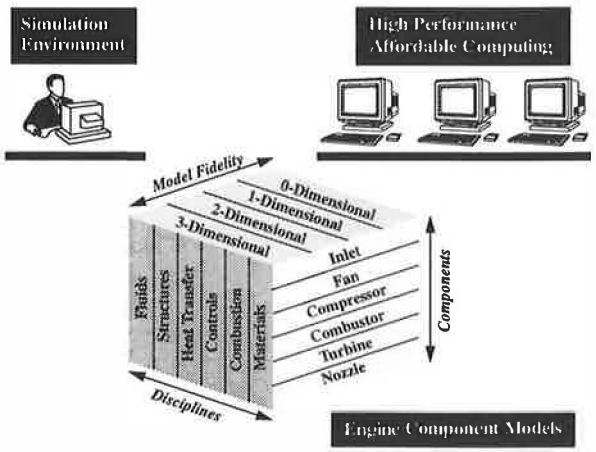


Figure 1: Topology of Computational Simulation System

The topology of such as system is shown in Figure 1. In this system, the engine component models are integrated to couple relevant disciplines, such as aerodynamics, structures, heat transfer, combustion, controls and materials. These models are then integrated at a desired level of fidelity (0-, 1-, 2-, or 3-dimensional) to form coupled subsystems for systems analysis. For required computing speed at a reasonable cost, simulation models can be distributed across a networked computing platform consisting of a variety of architectures and operating systems, including distributed heterogeneous parallel processors. A simulation environment provides a user-friendly interface between the analyst and the multitude of complex software packages and computing systems that form the simulation system.

The implementation of such as system is a major challenge. In this paper, we focus only on the design and development of a prototype *simulation environment* being developed in NPSS-related research.

1.1 Design Requirements

This section describes some of the high-level requirements which the gas turbine simulation environment must meet:

- **Component Level Modeling**. The primary requirement is for a platform which provides a general and flexible component view of the engine. Conceptually, this approach allows an engineer to develop new and different engine simulation models independent of the number of components in the engine, their type, fidelity level, or even location in the network.

- **Customization**. The environment must allow the user to customize simulation functionality by allowing components to be replaced by other components having different functionality. Such "pluggability" is essential for keeping the architecture current. Similarly, it must be capable of supporting the integration of new simulation techniques and computing methodologies with as little effort as possible. Specifically, this intended to address the areas of *multidisciplinary coupling* and *multimodeling*.

- **Component Interoperability**. In order for the preceding design requirements to be possible, it is essential that the user be guaranteed compatibility between all components used to develop simulation models. Component interoperability is enforced through specifications of a general component model.

- **Distributed Connection and Data Transformation**. The introduction of interdisciplinary models and multimodels requires support for distributed computing as it cannot be assumed that the higher-fidelity software will run efficiently (or at all) on the same computer platform as the rest of the system. Additionally, data transferred between components having different fidelity levels and/or data formats must be transformed accordingly.

- **Portability**. The environment must be capable of operating without regard to hardware or operating system combinations. This includes the ability to leverage extensive amounts of Fortran, C and C++ legacy software.

- **User Interface**. Finally, the system must provide a user interface to reduce the efforts of developing new models and executing simulations.

1.2 Alternatives

A great number of engine simulation software packages are currently in use. Most of these are proprietary software, developed and maintained by aerospace companies. Also a number of public domain software packages, developed by NASA and Universities are also in use [4, 5]. One approach is to determine the "best" software and modify it to address the design requirements listed above. However, it has been ours and others experience that this approach often requires more effort and produces less desirable results than a completely new design [6, 7]. Generally, this is due to the following [8]:

- *Procedural design structures.* Existing (public domain) simulation software tend to utilize global data structures, such as FORTRAN common blocks, to improve simulation execution times. However, the result is a lack of data encapsulation and safety, making changes in design difficult and dangerous.

- *Discipline isolation.* Most present-day simulation models offer only simplified coupling of interactions between, for instance, aerodynamics, structures and controls. Consequently, the design of the system reflect the bias towards these disciplines. This makes it difficult to introduce new models to couple additional disciplines.

- *Assumed single processor/machine environment.* Most simulation systems were designed not to exceed perceived implementation limitations (hardware, operating systems, memory, etc.). As a result,

the software's design impedes transport to modern parallel and distributed computing platforms.

- *No Graphical Interface.* Another drawback of most presently used simulation software is the lack of graphical user interfaces (GUIs). Engine models are developed by hand through input lists. This is often a tedious process which can also result in erroneous model definition.

An alternative approach which directly addresses the first limitation, and indirectly addresses the remaining limitations is to apply *object-oriented technology* to the development of the simulation environment. Object-oriented technology is a collection of powerful design, analysis and programming methodologies for creating general-purpose adaptive models and robust, flexible software systems [9, 10].

An object-oriented (OO) approach is attractive for modeling gas turbine systems due to the natural one-to-one correspondence between objects in the application and computational domains. Consequently, multifidelity and multidisciplinary representations of engine components can be conveniently encapsulated through the use of objects. Object class morphology provides the necessary structure to accommodate a common engineering model, and to define the essential interfaces for component and disciplinary coupling. Inheritance of methods and variables in the hierarchy of classes allows extension and customization of simulation models with an economy of effort. Moreover, the same structure can be used in the design, analysis, simulation and maintenance phases of the engineering cycle.

Several prototyping efforts to develop object-oriented gas turbine simulation systems have already been completed. The first was developed by Holt and Phillips [11]. In this work an object-oriented simulator was developed in Common Lisp Object System (CLOS) with component models based on the dynamic engine software package called DIGTEM [4]. Similar prototyping efforts were carried out by Curlett and Felder [6] and Reed and Afjeh [12] in C++ and Java™ programming languages, respectively. Results from these efforts were very encouraging. In particular, the use of a graphical user interface in both the CLOS and Java simulation software greatly increased flexibility in developing engine simulation models.

1.3 Solution to Design Challenge

These prototyping efforts have illustrated the flexibility and reusability of the object-oriented approach. However, this has been achieved mainly at the application level. In order to develop a next-generation simulation environment, we need to apply OO design concepts to the entire architecture. In recent years, two complementary concepts, *design patterns* and OO *application frameworks*, have been shown to be beneficial to developing reusable and flexible domain-specific software systems.

A framework is a reusable, "semi-complete" application that can be specialized to produce custom applications [13]. In general, the gas turbine simulation software now used are estimated to be ~80% identical, with the remaining percentage due to proprietary modifications of individual components. The great amount of commonality suggests that it might be possible to develop a generalized simulation software package based on the 80% of common features, and allowing the end user to customize the remaining 20% as desired. Frameworks can provide the necessary infrastructure to develop and mange such a generalized propulsion simulation system.

The benefits of object-oriented frameworks are due to their modularity, reusability, and extensibility. Frameworks enhance modularity by encapsulating volatile implementation details behind stable interfaces, thus localizing the impact of design and implementation changes [14]. These interfaces facilitate the structuring of complex systems into manageable software pieces — object-based *components* — which can be developed and combined dynamically to build simulation applications or composite components. Coupled with graphical environments, they permit visual manipulation for rapid assembly or modification of simulation models with minimal effort. Software component modularity also permits placement across computer platforms, making them well-suited for developing distributed simulations. Reuse of framework components can yield substantial improvements in model development and interoperability, as well as quality and performance of the computational simulation system. Frameworks enhance extensibility by providing "hooks" into the

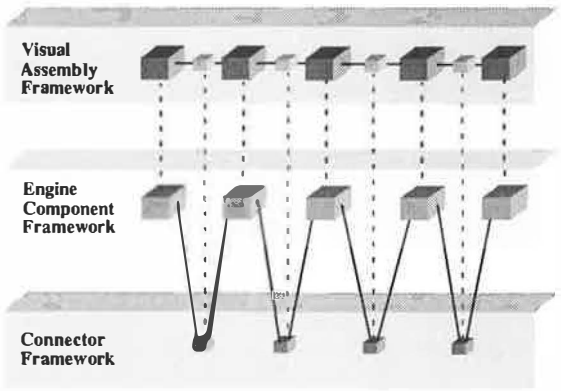


Figure 2: Onyx Framework Layers

framework. Coupled with the stable interfaces, these hooks allow the engineer to "plug" new functionality into the framework as desired. This is essential for keeping the simulation architecture current and facilitating new analytical approaches.

In the next section, we describe the design and implementation of *Onyx*, an object-oriented framework for aerospace propulsion simulation.

2 Overview of Onyx Framework

Onyx is structured as a *framework of frameworks*. Figure 2 illustrates the major structural frameworks and components which are described in this paper.

- *Engine Component Framework* - A database containing collections of domain-specific component models, such as a compressor and turbines, for use within the Visual Assembly Framework.

- *Visual Assembly Framework* - The Onyx graphical user interface (GUI) provides interactive control over the execution of the framework. The Visual Assembly Framework forms one part of the Onyx GUI and provides tools to visually assemble and manipulate a simulation model.

- *Connector Framework* - Provides a layered abstraction mechanism for distributed interconnection services between component models. Connectors also are utilized in interdisciplinary and multilevel component connections.

Onyx was developed using the Java object-oriented programming language and run-time platform [15]. Java was chosen for the Onyx framework because of its excellent object-oriented programming capabilities; platform-independent code execution (made possible through the use of byte-codes and a Java Virtual Machine); free availability on all major computing platforms; and, highly-integrated run-time class libraries, which serve as the foundation for Onyx's graphical user interface, distributed computing architecture, as well as providing future implementation of database and native code interfacing.

To illustrate how Onyx can be used to develop gas turbine simulations, we will present a running example throughout this section. A transient, lumped-parameter, aero-thermodynamic turbojet engine component model, developed in our previous research, is integrated within the framework. The resulting simulation system is capable of performing steady-state and transient analyses of arbitrarily configured jet engine models.

3 Engine Component Framework

3.1 Design Challenges

A gas turbine engine is essentially an assembly of *engine components* — inlet, fan, compressors, combustor, turbine, shafts and nozzle, etc. (see Figure 3a). These components operate together to produce power (or thrust). Engine components are themselves made up of other substructures. For example, a fan component may be expressed as a collection of hub, stage, casing, splitter and flowfield substructures (see Figure 3b). These in turn may be further decomposed into more basic elements such as rotor and stator blades.

Onyx's Engine Component Framework should allow users to simulate these structures at the various levels of abstraction as desired. For example, a user should be able to construct an engine component model from more basic models, and then use that component model to

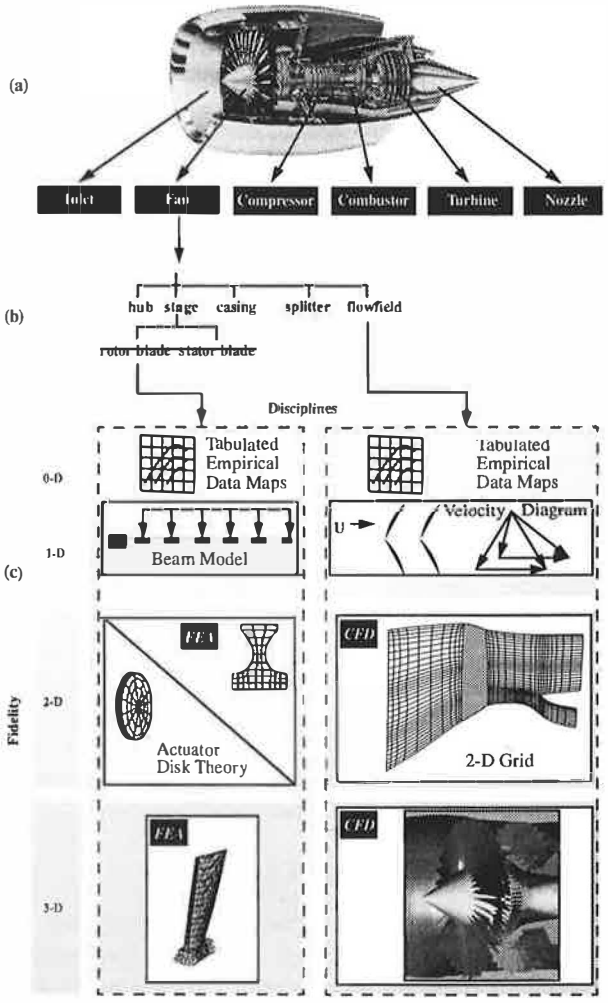


Figure 3: Engine Component Abstraction Diagram

build an engine model. Such an approach makes the process of developing gas turbine simulation models both simple and intuitive. To achieve this, we have selected an internal class structure which closely resembles the physical structure of the domain.

The component models internal structure should:

- maintain a component models physical relationship. This includes arrangement of any substructures as well as references to connected component models.

- provide control over the execution of the components simulation algorithm, which we call, its *analysis model.*

In developing the component model structure, we should not have to distinguish between single elements and assemblies of elements in our internal representation. For example, we should be able to treat a single rotor blade in the same manner as a fan component comprised of several elements, thus allowing the construction of arbitrarily complex models.

We can represent the hierarchal structure of the engine, its components, and substructures using *recursive composition.* This techniques allows us to build increasing complex elements out of simpler ones. Returning to Figure 3b, we can combine multiple sets of rotor and stator blades to form a fan component. The fan component can then be combined with other component-level elements (compressor, combustor, etc.) to form an engine model.

3.2 Engine Component Implementation

Figure 4 illustrates the structure of the Engine Components Framework in Onyx. For simplicity, only the more important variables and methods in the classes are shown. The structure of these classes is based mainly on the *Composite* design pattern [16]. This pattern effectively captures the part-whole hierarchal structure of our component models.

EngElement is a Java interface which establishes the common behavior for all engine component classes incorporated into Onyx. It defines the basic methods needed to initialize, run and stop engine element execution, as well as methods for managing Port objects. The abstract class DefaultEngElement implements EngElement and provides default functionality for the interface methods. In most cases, users will subclass DefaultEngElement to create concrete engine component classes, such as class XyzEngElement, to implement the

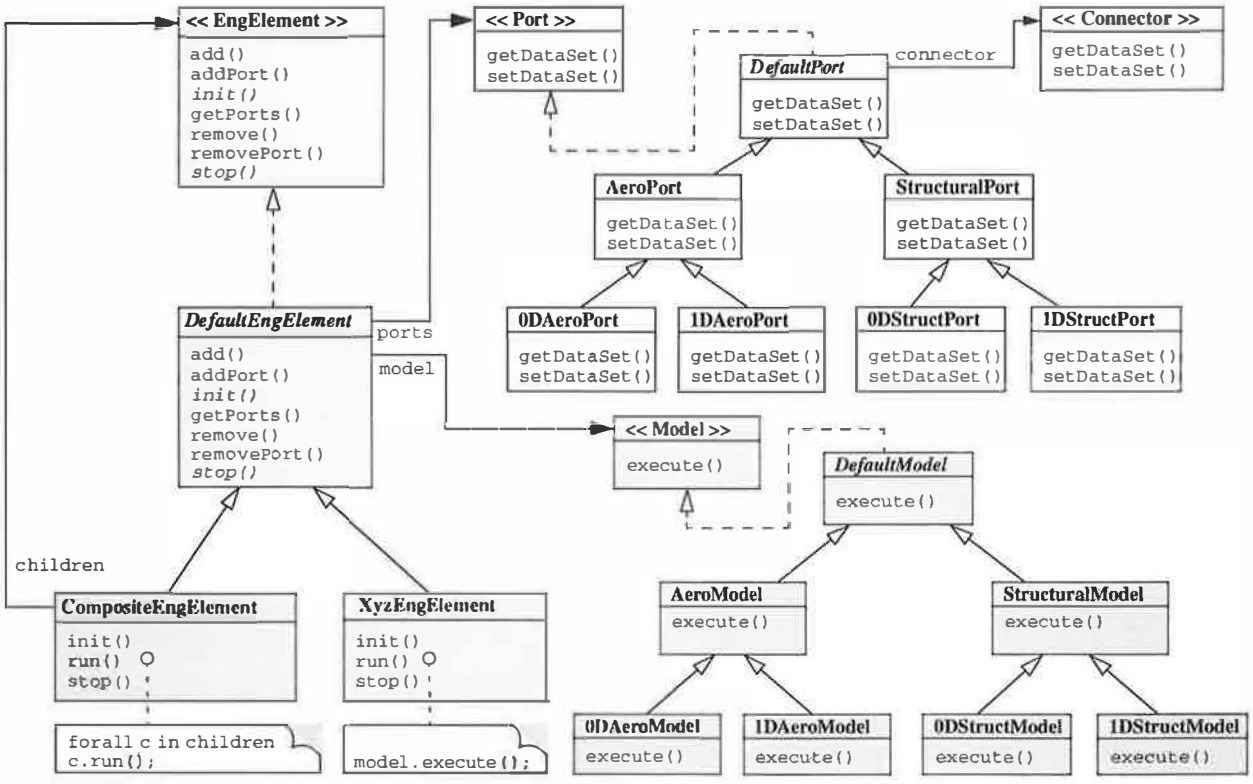


Figure 4: Structure of Engine Component Framework

required functional methods. The approach of providing a default abstract class for a Java interface is used throughout the Onyx system to give the user more flexibility when plugging in new classes. In this case, the user may select to inherit the functionality provided by DefaultEngElement, or to inherit from another class and implement the methods defined by the EngElement interface.

CompositeEngElement represents a composition of EngElement objects. Management operations for children are declared in DefaultEngElement to maximize component transparency. To ensure type-safety, these methods throw an exception for illegal operations, such as attempting to add or remove an EngElement from another EngElement, rather than a CompositeEngElement.

3.3 Analysis Model Implementation

Computational simulation involves designing a model of an actual or theoretical physical system, executing the model on digital computer, and analyzing the execution output [17]. Models are generally developed by defining a given problem domain, reducing the physical entities and phenomena in that domain to idealized form based on a desired level of abstraction, and formulating a mathematical model through the application of conservative laws.

Simulating complex systems requires the development of a hierarchy of models, or *multimodel*, which represent the system at differing levels of abstraction [18]. Selection of a particular model is based on a number of (possibly conflicting) criteria, including the level of detail needed, the objective of the simulation, the available knowledge, and given resources. For preliminary gas turbine engine design, simulation models are often used to determine the thrust, fuel consumption rates, and range of an engine. These simulations generally use relatively simple one-dimensional component models to predict performance. However, in other situations, such as *multidisciplinary analysis*, higher-order models are needed. For example, to prevent the possibility of a fan blade rubbing the cowling, an engineer might perform a coupled aerodynamic, thermal and structural analysis of the blade to determine the amount of blade bending due to the thermal and aerodynamic loading. Such an analysis would require several high-fidelity analysis models using fully three-dimensional, Navier-Stokes computational fluid dynamics (CFD) and structural Finite Element analysis (FEA) algorithms.

Ideally, one would prefer using three-dimensional analysis for an entire engine as it provides greater detail of the physical processes occurring in the system. The computational requirements for such an analysis, however, far exceed present computer capabilities. Consequently, it is desirable for an EngElement to be capable of accommodating views having multiple levels of fidelity and differing disciplines. Figure 3c illustrates the concept of multiple views for a rotor blade and flowfield objects in a fan component. The rotor blade is analyzed using various mechanical-structural methods, while the flowfield is represented by various aero-fluid-dynamic methods. Based on the simulation criteria, an appropriate analysis model may be selected.

The complexity of the various analysis models suggest that it is desirable to encapsulate the analysis model, or remove it from the structure of EngElement. This would protect the modularity of EngElement, allowing new EngElement classes to be added without regard to the analysis model, and conversely to add new analysis models without affecting the EngElement class.

We apply the *Strategy* design pattern [16] to encapsulate the analysis model in an object. The DefaultModel class is an abstract class which implements the Model interface. The interface defines the methods which all Models must support to be integrated within Onyx. As an example, two analysis models, 0DAeroModel and 1DAeroModel, are shown as subclasses of AeroModel.

3.4 Ports

Completing the Engine Component Framework structure is the Port class. In physical terms, a Port represents a control surface through which energy and mass flow between engine components. In Onyx, Ports define an interface between EngElements through which data is passed. Port is an abstract class which defines the default functionality, and maintains a reference to a Connector. Connectors will be discussed in section 5. Port is subclassed according to the discipline (e.g. aerodynamic, structural, thermal, etc.), and these classes are then each subclassed by fidelity (0-D, 1-D, 2-D, 3-D). Which subclass of Port an EngElement instantiates is determined by the discipline-fidelity combination of the EngElements analysis model(s). For example, if EngElement has a single analysis model which is a 0-D, aerodynamic model, then an instance of EngElement creates two 0DAeroPort objects to handle input and output. Because the analysis model is dynamic and may be changed at run-time, the Port objects also must change accordingly. Consequently, we apply the *State* design pattern [16] to dynamically create and manage the Ports in an EngElement.

3.5 Example

. To illustrate the application of the Onyx framework and the feasibility of this approach, a small collection of

component object classes representing the inlet, compressor, combustor, turbine, nozzle, bleed-duct connecting-duct, and shaft, of a jet engine have been developed. An inter-component mixing volume class was also defined which is used to connect two successive components as well as define temperature and pressure at component boundaries. These concrete classes are all subclasses of the abstract DefaultEngElement class shown in Figure 4.

Each class implements a specific mathematical (analysis) model which describes its physical operation. In this example, the analysis models are all relatively simple differential-algebraic equations (DAE) developed from an space-averaged treatment of the conservative laws of thermo- and fluid dynamics. These are patterned after the work of Daniele et al., [4]. A complete description of the models can be found in the work of Reed [7]. The analysis model for each component is encapsulated in an appropriate subclass of DefaultModel, and present specific implementations of the `init()`, `run()` and `stop()` methods which initialize the component and execute its analysis model, respectively.

Appropriate Port objects are created in each component object depending on the number and type of connections required. For example, a compressor class defines two AeroPort objects to pass aero-thermodynamic data to adjoining components, and a StructuralPort to pass data to a connecting shaft object.

4 Visual Assembly Framework

4.1 Design Challenges

Aerospace engineers often use schematic drawings to represent propulsion systems and subsystems. It is then natural to represent computational simulations of such systems using this visual metaphor.

In the previous section, we developed an object-oriented component model which allows us to dynamically assemble arbitrarily complex engine system models. We now consider the development of a framework which supports visual assembly of those component models.

The main requirement of the Visual Assembly Framework is to provide visual analogs for the component model objects, and support for assembling them. This has several implications. The first is obvious: we need visual elements to represent the objects which form Onyx's engine component model. The second, less obvious requirement, is that the concept of component composition developed previously must also be supported visually. Finally, the framework must take care of managing basic graphical functions — window management, displaying objects, moving and dragging visual elements, tracking mouse movements, etc. This reduces the programming burden for engineers using the framework.

In addition to these goals are some constraints. First, the framework should decouple the visual user interface (UI) objects from their counterparts in the component framework. Although the visual elements represent the component, we would like to allow a component's UI to be changed easily, possibly at run-time.

Second, our implementation should allow the user to override the default visual representations as much as is practically possible.

We have selected the Java platform in part because of its integrated graphical support. Java's Abstract Window Toolkit (AWT) is part of the core classes which are available in every Java Virtual Machine (JVM). The AWT provides a collection of platform-independent graphical components for building graphical applications in Java. One drawback of the AWT is that it provides only basic low-level graphical components. Another drawback is the heavyweight nature of the AWT, due to implementing graphical objects with the native windowing system.

We have opted instead to use the Swing component set to implement our graphic interface [19]. Swing is a subset of the new Java Foundation Classes (JFC), which is itself a subclass of the AWT. Therefore, our graphical interface will retain the same portability made possible with AWT. Swing however, adds more high-level graphic components, as well as the ability to select from multiple Look-and-Feel standards. However, this selection raises some immediate implementation issues.

One attractive feature of Java is its capability to develop *applets* — compiled Java programs which can be dynamically downloaded from a Web server and run locally on the client's machine using a Java-enabled browser. The ubiquity of Web browsers make implementing Onyx's visual assembly framework as an applet very attractive.

One drawback of using an applet is the relatively long time needed to implement new versions of the JVM into web browsers. Currently, the JFC is not implemented in any browser, meaning that the Swing classes used in Onyx would have to be downloaded along with the visual assembly framework each time the applet was accessed.

Another drawback associated with using an applet is its security restrictions which affect the partitioning of Onyx's structure. Generally, this limits communications between the applet to only the web server from which it was downloaded.

Because of these issues, we have designed the visual assembly framework as a Java *application*. Applications

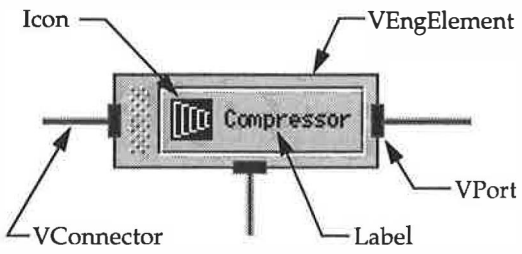


Figure 5: SchematicIcon

are similar to stand-alone programs. As the issues of browser-JVM integration and applet security issues are addressed, we will modify Onyx to permit the visual assembly framework to be distributed as an applet.

4.2 Visual Assembly Framework Design

A simulation model is constructed by creating SchematicIcon objects and connecting them to form an engine schematic. A SchematicIcon is composed of a VEngElement and one or more VPorts. VConnectors are used to "wire" the SchematicIcons together. Figure 5 illustrates these relationships.

- VEngElement is the visual analog of the EngElement class in the component framework. VEngElement is a subclass of `java.swing.JButton`, and thus contains an Icon which presents an image of the engine component; a Label which displays the name of the EngElement object instance.

- One or more VPorts are attached to the VEngElement, and represent connection points between components. VPorts are color-coded to represent the type of Port it represents.

- A VConnector is the visual analog of the Connector object. It is represented as a line drawn between two VPorts.

Each VEngElement, VPort and VConnector has a popup menu associated with it. The menu allows the user to access various functions such as moving, deleting, copying, etc. In the VEngElement, the popup menu has a special item for "customizing" the component. When selected, the customizer object is displayed.

Customizers are graphical interfaces which allow the user to change an EngElement's attributes. Typically, these are used to modify data in the EngElement analysis model. They may also be used to control the distribution of the EngElement in a distributed simulation.

In designing the structure for our visual assembly we immediately recognize from Figure 5 that each instance of SchematicIcon represented in the framework will likely have different Icons, display names and VPorts. One solution is to define SchematicIcon as an abstract class, and use inheritance to define subclasses which represents visually the various concrete SchematicIcon classes. Each class would then redefine the Icon image, display name, and VPort location and type. This approach however, typically leads to a very broad and shallow inheritance tree, indicating little use of inheritance.

A more useful approach would be to create an appropriate SchematicIcon using object composition. This is accomplished through the use of the parameterized *Factory* design pattern [16], in conjunction with Java's reflection mechanism. This also allows us to address one of the design constraints listed previously: decoupling a component's UI from its component model representation. Our solution is to apply a variation of the JavaBeans™ "Info" class concept [20].

We will illustrate this approach by creating a SchematicIcon object for an XyzEngElement object (see Figure 6). When a user creates an instance of XyzEngElement (this process will be discussed later), the Visual Assembly Framework invokes the SchematicIconFactory's `create()` method. This method invokes the `getEngElemInfo()` method in the XyzEngElement object which returns the info class name, `XyzEngElementInfo.class`. The Factory instantiates this class using the `java.lang.reflect.Constructor newInstance()` method. XyzEngElementInfo implements the EngElementInfo interface which defines two methods to create and return instances of PortDescriptor and EngElementDescriptor. We create and return instances

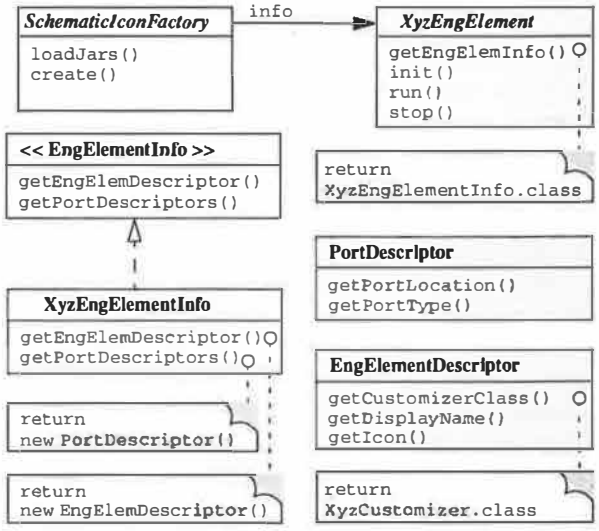


Figure 6: SchematicIcon creation process

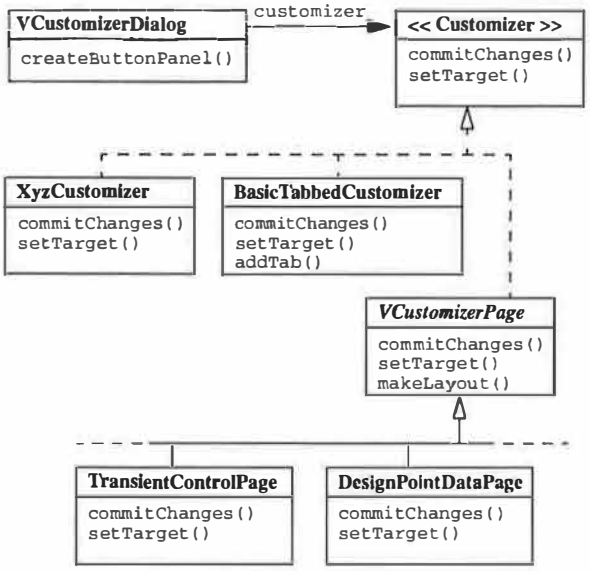


Figure 7: Customizer structure

of these classes instead of simply returning the class name, since XyzEngElementInfo initializes these instances by passing parameters in the constructor of each class.

PortDescriptor encapsulates information concerning the type, initial placement, and constraints of the VPorts for XyzEngElement. EngElementDescriptor defines methods which return the Icon image and display name String used in the VEngElement button. The method `getCustomizerClass()` returns the class name for the XyzEngElement's Customizer. This class name is stored in the VEngElement object, and is lazily initialized using the `newInstance()` method.

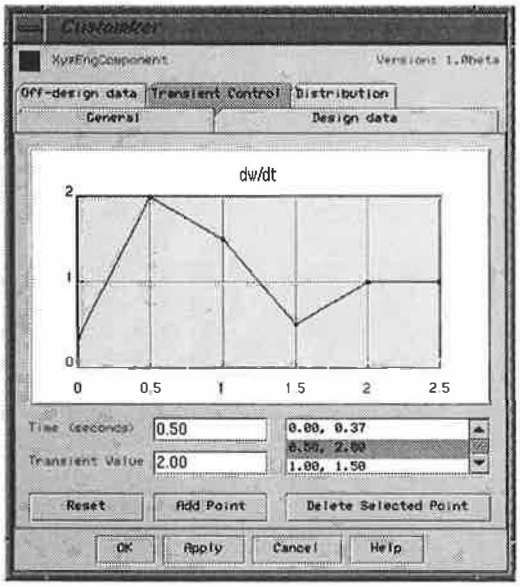


Figure 8: Onyx Customizer

The combination of the *Factory* pattern and Java reflection gives considerable freedom and flexibility in creating SchematicIcons. The composition of a SchematicIcon can easily be redefined by subclassing SchematicIconFactory. We can also use Java reflection to alter the specific classes that get instantiated in order to build the SchematicIcon without subclassing. Furthermore, we have effectively separated the UI implementation from component implementation. One drawback to this approach is the level of indirection introduced. However, the user sees little of this complexity as he or she is only required to define the XyzEngElement, XyzEngElementInfo and a Customizer class.

4.2.1 Customizers

We face another dilemma in creating customizers for each EngElement. Customizer represents a UI for defining and editing the attributes of an EngElement's analysis model. Because it is strongly coupled to the data structure for each specific type of EngElement, we will likely end up with many different Customizer classes. These may or may not have any commonality, so we may not be able to take advantage of inheritance. In order to be flexible, Onyx must be capable of integrating each of these specific Customizers. Furthermore, we would like to allow users as much flexibility as is possible to customize the data UI, so we do not want to limit their options through inheritance.

Our solution is to provide an interface which defines a *plug-point* for user-defined customizers. Figure 7 shows the Customizer structure. To maximize flexibility, the Visual Assembly Framework allows the user to 1) program a new customizer, or 2) to use the BasicTabbedCustomizer. A user-defined customizer would inherit from `java.awt.Component` and implement the Customizer interface methods directly. The `commitChanges()` and `setTarget()` methods are called from the Visual Assembly framework. The constraint of inheriting from `Component` is necessary as all customizers are automatically added to an instance of VCustomizerDialog which expects its child to be a subclass of `Component`. VCustomizerDialog wraps the Customizer and provides a set of buttons to accept user input. The `setTarget()` method identifies the object to be updated, while the `commitChanges()` method is used to update the object when the user accepts changes to the customizer data. XyzCustomizer is an example of a user-defined customizer.

In the second approach, the user can subclass VCustomizerPage, compose it with the desired UI objects, and add it to BasicTabbedCustomizer. VCustomizerPage can provide methods to handle common issues

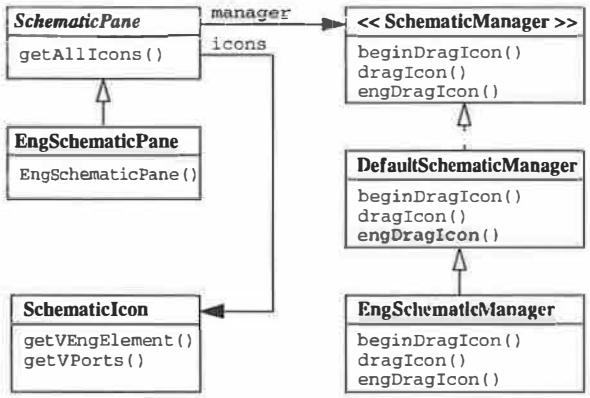


Figure 9: SchematicFrame structure

such as laying out components. Since BasicTabbedCustomizer adds instances of Customizer, it is also possible to add classes which inherit from `java.awt.Component` and implement Customizer. Figure 8 shows a picture of an instance of DefaultTabbedCustomizer, including several VCustomizerPage page objects.

The Customizer structure provides considerable flexibility. It allows the user select to compose the UI or inherit functionality and structure when developing a customizer. By adhering to an interface, users can develop different customizers and plug them in as desired. This process is made relatively easy with a simple change to the class name returned by the `getCustomizerClass()` method in EngElementDescriptor. Furthermore, since the VCustomizerDialog accepts subclasses of `java.awt.Component`, users can use Java Integrated Development Environments (IDEs) to quickly construct customizers from AWT or Swing Java Bean GUI components.

4.2.2 Frames, Panes, Managers and SchematicIcons

Engine schematics are built by adding SchematicIcons to a EngSchematicPane which is contained in a SchematicFrame. EngSchematicPane is a subclass of `java.swing.JLayeredPane`, and maintains a listing of the SchematicIcons it contains, as well as their z-order (i.e., their layer). EngSchematicPane also keeps a reference to an EngSchematicManager, which provides support for selecting and moving SchematicIcons within the EngSchematicPane (see Figure 9).

The SchematicFrame and related classes provide required support for user interactions: dragging, moving, etc. We also support in the visual framework, the hierarchal composition concept introduced in the component framework.

In our requirements for a visual assembly framework, we indicated our desire to support visually the

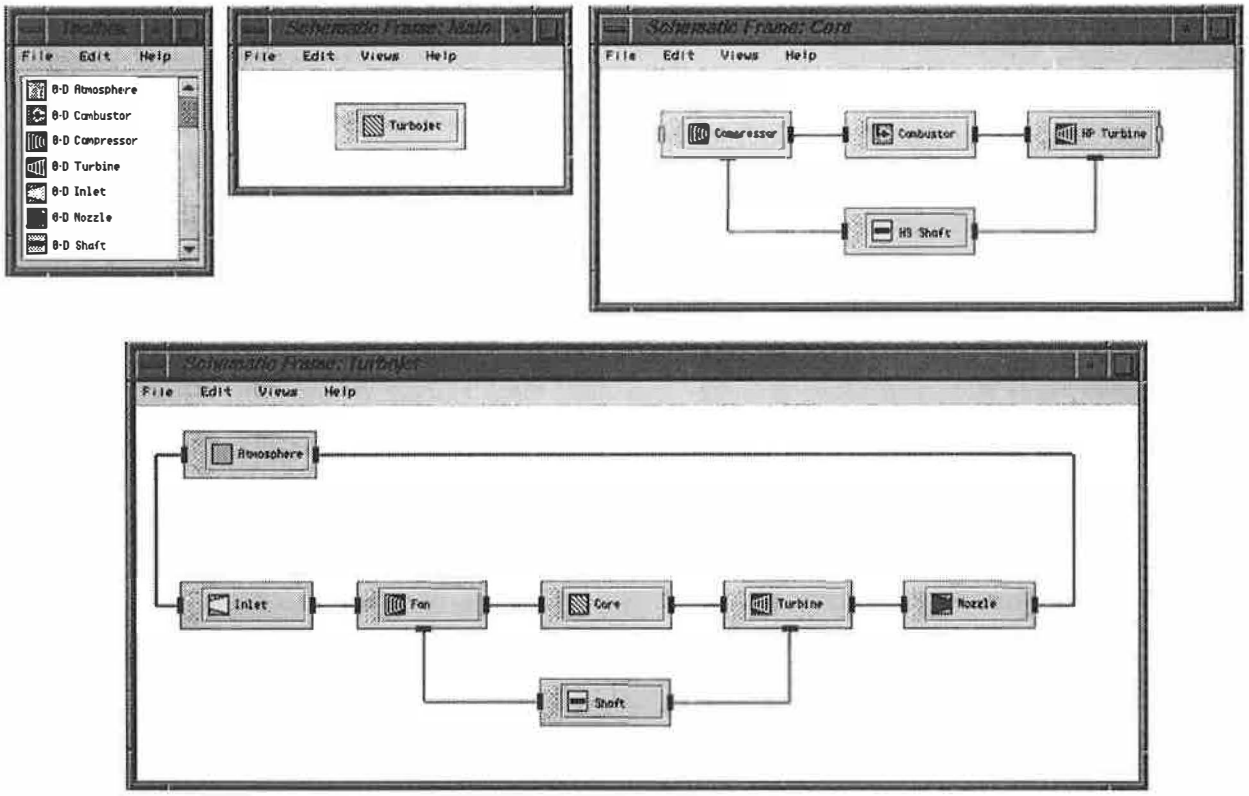


Figure 10: SchematicFrames showing visual composition

composition of EngElements in the Engine Component Framework. This is implemented using the SchematicFrame, EngSchematicPane and SchematicIcons classes. We illustrate it with an example (see Figure 10).

4.3 Example

In section 3.5, the EngElement classes representing engine components found in a turbojet engine were developed. We now demonstrate using the classes in the Visual Assembly Framework to create a simple turbojet engine. For each EngElement class, the user also defines an EngElementInfo class with appropriate descriptor information; including the icon, display name, customizer, and VPort locations. The customizer, EngElementInfo, and EngElement model and port classes for each EngElement are then collected into a Java archive (jar) file.

When the Visual Assembly Framework is started, Onyx searches the default loading directory, and loads the classes for each of the jar files. The EngElement classes are extracted and stored for instantiation by a factory object. The EngElementInfo classes are also extracted and used to obtain the display names and icons for each of the loaded EngElements. The icons and display names are listed in the Visual Assembly Toolbox which is displayed alongside the initial SchematicFrame, called `Main` (see Figure 10). From the `Main` window, the user selects the `Create Composite` menu command, which creates a new SchematicFrame and places a Composite SchematicIcon in the Main window. This SchematicIcon represents the top-level view of the turbojet engine, and the user names it `Turbojet`. This also sets the title name of the new SchematicFrame to `Turbojet`.

Next, the user begins to construct the turbojet engine model. From the Toolbox, the user selects an `Inlet`, `Fan`, `Shaft`, `Turbine` and `Nozzle` engine component to add to the `Main` SchematicFrame. This action creates an proper SchematicIcon for the each component and displays them in the EngSchematicPane. At the same time, Onyx instantiates their respective EngElements and adds them to an instance of CompositeEngElement in the Engine Component Framework. Our user next selects the `Main` SchematicFrame, and using the `Create Composite` command, instantiates a second SchematicFrame, which the user names `Core`.

From the Toolbox, the user now selects `Compressor`, `Combustor`, `Shaft` and `Turbine` components to add to the `Core`. SchematicIcons for these components are created and displayed in the `Core` EngSchematicPane. Onyx instantiates their respective EngElements and adds them to a second instance of CompositeEngElement in the Engine Component Framework. At this time, a SchematicIcon representing the `Core` SchematicFrame is added to the `Main` EngSchematicPane.

We now have two loosely coupled composite hierarchal structures: one composed of EngElements within CompositeEngElements in the Engine Component Framework; and its corresponding visual representation composed of SchematicIcons within EngSchematicPanes. Also notice, from Figure 10, that the relationships between SchematicIcons, VPorts and VConnectors are maintained in both the `Main` and `Core` frames.

5 Connector Framework

5.1 Design Challenges

We have developed a component model for gas turbine components, as well as a compatible interface so that they can be assembled — both programmatically and visually — to form more complex systems of objects. In order for these components to interact and simulate the given system, they need to communicate. In the Onyx architecture, EngElements communicate by sending messages via a Port.

Consider a physical connection between a Inlet and Fan EngElements as shown in Figure 10. Inlet and Fan are physically and logically connected and exchange messages, such as `getDataSet()`, to retrieve data in order to update their analysis models. Normally, this process would be relatively straightforward, with the `getDataSet()` request being forwarded from the Fan via the Fan's Port to the Inlet's Port, and finally to the Inlet, where the request is carried out. In the Onyx architecture, however, this process is made more complicated by at least two situations.

5.1.1 Multifidelity Connections

The first situation occurs when two EngElements are connected which have analysis models with different discipline and/or fidelity combinations. If, for example, the Inlet component has a 1-D Fluid model and the Fan has a 2-D Fluid model, then we have a mismatch in fidelity. When the Fan processes the `getDataSet()` message, it would have some intelligence capable of transforming its 2-D data into a 1-D data set before returning it to the Inlet. Other methods are also needed to perform additional transformations (2-D to 0-D, 2-D to 3-D, etc.). Such transformation methods are clearly necessary in order for the Onyx architecture to support interdisciplinary and multifidelity modeling

Holt and Phillips [11] introduced the concept of *connector* objects to provide appropriate methods for "expanding" or "contracting" the data, and mapping

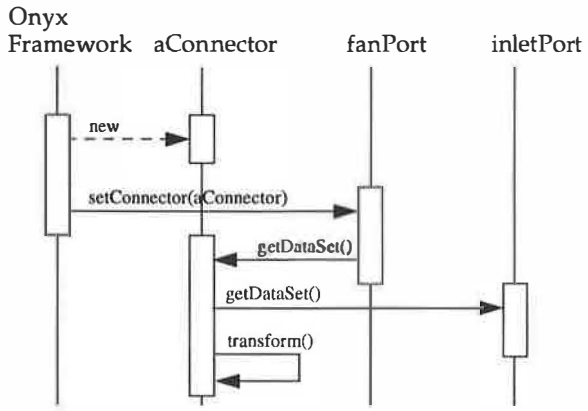


Fig. 11 - Interaction diagram

from different discipline domains. Connector objects are essentially intelligent Command objects, as described by the *Command* design pattern [16]. As with the command objects, connectors provide flexibility by decoupling the collaborating objects, making them easier to reuse. An EngElement no longer need know the discipline-fidelity of the EngElement to which it is connected. Figure 11 shows an interaction diagram using connectors.

5.1.2 Distributed Connections

The second problematic situation results from the fact that an EngElement is to be distributable to other machines. The complex and intensive computational nature of jet engine simulations require that the framework be capable of distributing computations on a network of computers. This permits access to high-performance mainframe or workstation clusters for computationally intensive tasks and, at the same time, permits user control from the local computer. Also, this feature allows on-line monitoring of computations and dynamic allocation of computational resources for optimum performance while a simulation is in progress.

Although the distribution of objects across a network is a relatively complex task, our goal is to design Onyx to perform this distribution in a manner totally consistent with non-distributed simulations. Consequently, the distribution of components across the network should be as transparent as possible to the user. No actions, other than selecting a remote machine on which to run a component, should be required to distribute the component at run-time. To illustrate the process, we return to our Inlet-Fan example.

In this scenario, the user would like to run the Fan component on a remote machine. In the Visual Assembly Framework, the user creates an Inlet and Fan. Accessing the Fan's customizer (see Figure 8), the user selects the "Distribution" page and selects from the list, the name of the remote machine on which to run the Fan. Now, the user (implicitly) creates a Connector by drawing a connecting line between the Inlet and Fan Ports. Notice, that with the exception of selecting the name of the remote machine, the process is exactly the same as connecting components which run on the same machine.

Placing a component object on the remote machine, however, means that the two components reside in different Java Virtual Machines. This raises a difficulty since a Connector has two variables, `port1` and `port2`, which keep references to the Port objects connected to the Connector. One of these variables would normally be referencing the Fan, but since it is in a different virtual machine, it cannot be referenced.

We can address this problem by having the Connector reference a *remote proxy*, as defined in the *Proxy* design pattern [16]. The remote proxy provides a local representation for an object in another design space.

5.2 Connector Framework Implementation

The Connector Framework structure is shown in Figure 12. The Java interface, Connector, defines our interface functionality. As with previous interfaces, we provide an abstract class, DefaultConnector, which implements the interface, provides default

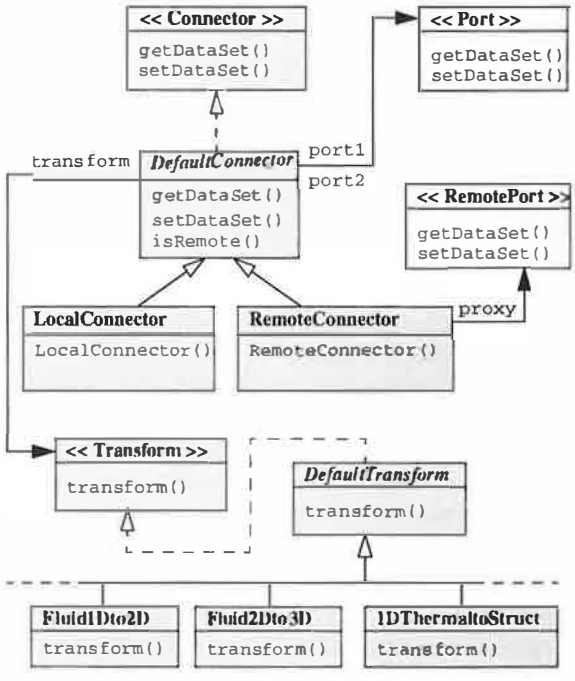


Figure 12: Structure of Connector Framework

implementation of each method, and defines the variables `port1`, `port2`, and `isRemote`. LocalConnector inherits all of its functionality and variables from its superclass. It represents a normal (non-remote) Connector. References to the `port1` and `port2` objects are passed into the constructor. RemoteConnector's constructor takes an additional argument to identify the remote machine. RemoteConnector defines a `proxy` variable to hold the reference to the proxy object. The constructor also initializes Onyx's interconnection service to bind `proxy` to the remote object.

Onyx's distribution mechanism is currently based on the Java Remote Method Invocation (RMI), a core component of the Java platform. RMI uses client stubs and server skeletons to interface with the local and remote objects. The stub represents the remote proxy object which is referenced by the RemoteConnector `proxy` variable

Because RMI is designed to operate fully within the Java environment, it is limited to connections between machines which are running the Java Virtual Machine. By assuming the homogeneous environment of the JVM, Onyx can take advantage of the Java object model whenever possible. This provides a simple and consistent programming model. Given that most computing platforms now provide a JVM, this should not limit the use of the framework. However, we are also in the progress of integrating CORBA for providing non-Java distributed object support. This is especially important for incorporation of the multitude of legacy applications not written in Java which currently exist in the aerospace industry.

Our Connector now provide two sets of functionality: 1) it can transform data sets between two components of different fidelity, and 2) it establishes and maintains communications between distributed components. Although both functions are based on decoupling the connected components, we would prefer that Connector has a more singular functionality. This would make it more reusable in the future. To achieve this, we delegate the transformation responsibility to a separate Transform object. Connector selects an appropriate Transform object using a *State* pattern [16], based on the fidelity-discipline combination of the connection. The Transform object utilizes the *Strategy* pattern [16], to allow different transformation algorithms, such as Fluid1Dto2D, to be interchangeable.

The Connector makes connections between EngElements transparent. Both distributed and multifidelity connections can be made without regard to location of the component, or its fidelity. Modifying the distribution mechanism can be performed either by subclassing DefaultConnector, or implementing Connector directly. Also, connection implementation details are fully encapsulated by the Connector, allowing EngElement and Port to remain unaffected by an changes to the distribution mechanism.

5.3 Example

For test purposes we have established a simple peer-to-peer distribution mechanism for the EngElement objects in our example model. EngElement objects are instantiated on the remote machine and export their interface so that their `init()`, `run()` and `stop()` methods may be called by Onyx from a local machine. In addition, a RemotePort interface was defined and is exported to allow connections from local (non-remote) Port objects. This interface allows the connectors and ports to invoke the getDataSet() methods to return a serialized object containing necessary engine component operating states.

Future efforts in this area will investigate the use of mobile object technology, such as ObjectSpace's Voyager [23], to allow the user to dynamically relocate EngElement objects to other platforms on the network.

6 Concluding Remarks

Designing and developing new aerospace propulsion technologies is a time-consuming and expensive process. Computational simulation is a promising means for alleviating this cost, due to the flexibility it provides for rapid and relatively inexpensive evaluation of alternative designs, and because it can be used to integrate multidisciplinary analysis earlier in the design process (Jameson, 1977). However, integrating advanced computational simulation analysis methods such as CFD and FEA into a computational simulation software system is a challenge. A prerequisite for the successful implementation of such a program is the development of an effective simulation framework for the representation of engine components, subcomponents and subassemblies. To promote concurrent engineering, the framework must be capable of housing multiple views of each component, including those views which may be of different fidelity or discipline (Irani et al., 1994). In addition, the framework must address the challenges of managing this complex, computationally intensive simulation in a distributed, heterogeneous computing environment.

Object-oriented application frameworks and design patterns help to enable the design and development of aerospace simulation systems by leveraging proven software design to produce a reusable component-based architecture which can be extended and customized to meet future application requirements. The Onyx

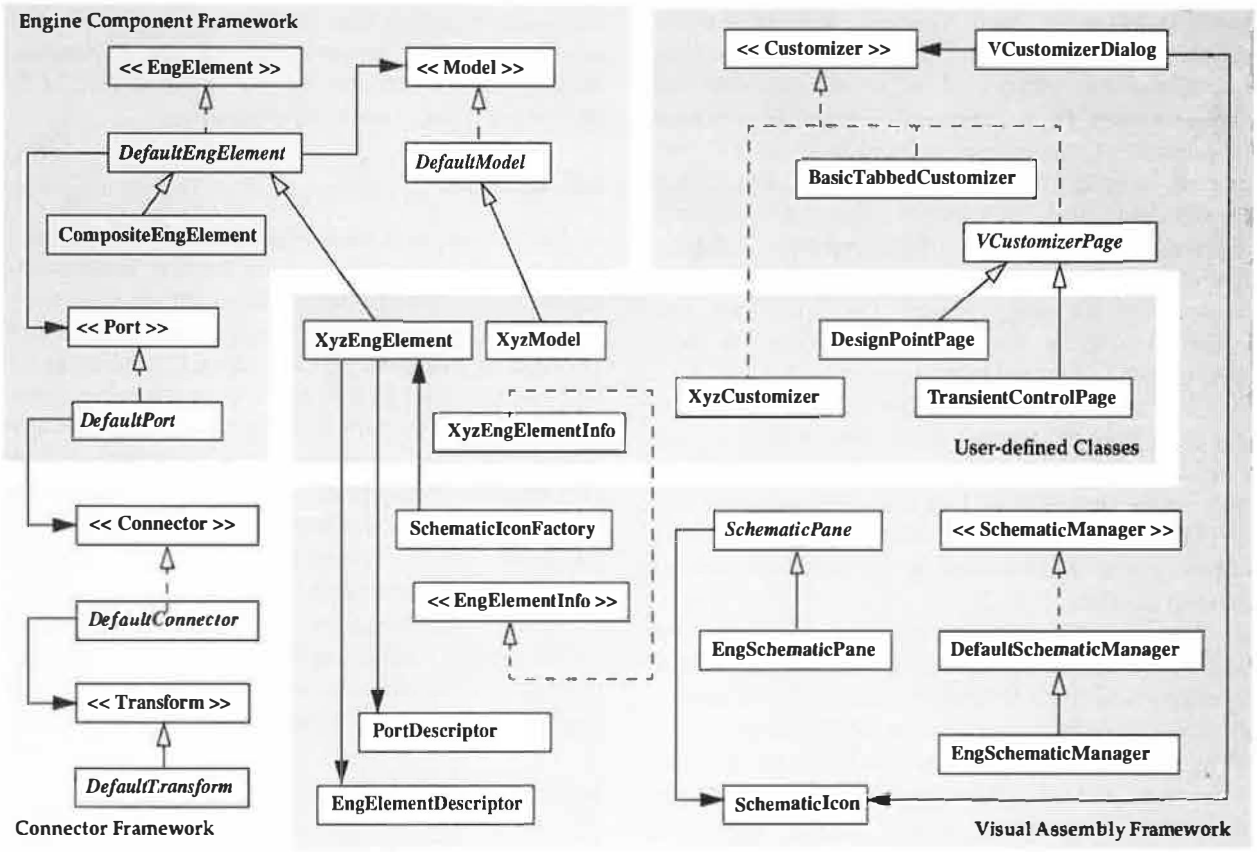


Figure 13: Onyx Aerospace Propulsion System Simulation Framework

application framework described in this paper provides an ensemble of framework components which, together, form an integrated framework for propulsion system simulation. Figure 13 shows how the individual framework component structures combine to form the Onyx framework.

Onyx promotes the construction of aerospace propulsion systems, such as jet gas turbine engines, in the following ways. First, it provides a common engine component object model which: encapsulates the hierarchal nature of the physical engine model, is capable of housing multimodel and multifidelity analysis models, and enforces component interoperability through a consistent interface between components. Second, it enables the construction and of engine models and customization of the simulation at a high level of abstraction through the use of visual representation in the visual assembly framework. Third, it supports both connection and transformation of data between multifidelity components running in a distributed network environment. Finally, the object-oriented design, built-in support for graphical interfaces and heterogeneous distributed processing, and automatic memory management, in Java greatly simplify and unify the design and development of Onyx. In addition, Java's byte code and widely available Java Virtual Machine allows Onyx to be highly portable.

The use of object-oriented application framework and design pattern methods in Onyx help to decouple domain-specific simulation strategies from their implementations. This decoupling enables new simulation strategies (e.g., components, analysis models, solvers, etc.) to be integrated easily into Onyx. By applying these design strategies, Onyx allows users to dynamically alter simulation models during any phase of the simulation. The example presented in the paper serves to illustrate the flexibility, extensibility, and ease of using Onyx to develop aerospace propulsion system simulations.

Acknowledgments

The work described in this paper was made possible by funding from the NASA Lewis Research Center Computing and Interdisciplinary System Office (Grant No. NCC-3-207), and the University of Toledo. Mr. Reed is partially supported by a University of Toledo Doctoral Fellowship. We would like to thank Greg

Follen at NASA Lewis for his continued support. Special thanks also to Murthy Devarakonda for his guidance in preparing this paper. More information on Onyx is available at www-mime.eng.utoledo.edu/~jreed/.

References

[1] Claus, R. W., Evans, A. L., Lytle, J. K., and Nichols, L. D., 1991, "Numerical Propulsion System Simulation," *Computing Systems in Engineering,* Vol. 2, pp. 357-364.

[2] Claus, R. W., Evans, A. L., and Follen, G. J., 1992, "Multidisciplinary Propulsion Simulation using NPSS," AIAA Paper No. 92-4709

[3] Evans, A. L., Lytle, J., Follen, G., and Lopez, I., "An Integrated Computing and Interdisciplinary Systems Approach to Aeropropulsion Simulation," ASME Paper No. 97-GT-303.

[4] Daniele, C. J., Krosel, S. M., Szuch, J. R., and Westerkamp, E. J., 1983, "Digital Computer Program for Generating Dynamic Engine Models (DIGTEM)," NASA TM-83446.

[5] Plencer, R., and Snyder, C., 1991, "The Navy/NASA Engine Program (NNEP89) - A User's Manual," NASA TM-105186.

[6] Curlett, B. P. and Felder, J. L., 1995, "Object-oriented Approach for Gas Turbine Engine Simulation," NASA TM-106970.

[7] Reed, J. A., 1993, "Development of an Interactive Graphical Aircraft Propulsion System Simulator," MS Thesis, The University of Toledo, Toledo, OH.

[8] Drummond, C., Follen, G., and Cannon, M., 1994, "Object-Oriented Technology for Compressor Simulation," AIAA Paper No. 94-3095.

[9] Taylor, D. A., 1990, "Object-Oriented Technology: A Manager's Guide," Addison Wesley Publishing Company, Inc., Reading, MA.

[10] Booch, G, 1991, "Object Oriented Design with Applications," The Benjamin/Cummings Publishing Company, Inc., New York, NY.

[11] Holt, G., and Phillips, R., 1991, "Object-Oriented Programming in NPSS Phase II Report," NASA CR-NAS3-25951.

[12] Reed, J. A., and Afjeh, A. A., 1997, "A Java-based Interactive Graphical Gas Turbine Propulsion System Simulator," AIAA Paper No. 97-0233.

[13] Johnson, R. and Foote, B., 1988, "Designing Reusable Classes," *Journal of Object-Oriented Programing*, Vol. 1, pp. 22-35.

[14] Schmidt, D. C., 1997, "Applying Design Patterns and Frameworks to Develop Object-Oriented Communications Software," *Handbook of Programming Languages, Volume I*, P. Salus, ed., MacMillian Computer Publishing.

[15] Arnold, K. and Gosling, J., 1996, "The Java Programming Language," Addison Wesley Publishing Company, Inc., Reading, MA.

[16] Gamma, E., Helm, R, Johnson, R., and Vlissides, J., 1995, "Design Patterns: Elements of Reusable Object-Oriented Software," Addison Wesley Publishing Company, Inc., Reading, MA.

[17] Fishwick, P. A., 1997, "Computer Simulation: Growth Through Extension," *TRANSACTIONS of The SCS*, Vol. 14, pp. 13-23.

[18] Fishwick P. A. and Zeigler, B. P., 1992, "A Multimodel Methodology for Qualitative Model Engineering," *ACM Transactions on Modeling and Computer Simulation*, Vol. 12, pp. 52-81.

[19] Swing, 1997, "The Swing Connection," Available from http://java.sun.com/products/jfc/swingdoc-current/index.html.

[20] Englander, R., 1997, "Developing Java Beans," O'Reilly & Associates, Inc., Sebastopol, CA.

[21] Jameson, A., 1997, "Re-Engineering the Design Process through Computation," AIAA Paper No. 97-0641.

[22] Irani, R. K., Graichen, C. M., Finnigan, P. M., and Sagendorph, F., 1994, "Object-based Representation for Multidisciplinary Analysis," AIAA Paper No. 94-3093.

[23] Voyager, 1998, "ObjectSpace Voyager Core Package Technical Overview," Available from http://www.objectspace.com/.

Building a Scalable and Efficient Component Oriented System using CORBA – Active Badge System Case Study

Jakub Szymaszek, Andrzej Uszok and Krzysztof Zieliński
Institute of Computer Science
University of Mining and Metallurgy (AGH)
Al. Mickiewicza 30, 30-059 Kraków, Poland
{jasz,uszok,kz}@ics.agh.edu.pl, http://galaxy.uci.agh.edu.pl~{jasz,uszok,kz}

Abstract

This paper presents experienc gathered when implementing the localization system for an office environment in CORBA. It describes methods which enable preserving fine-grained object-oriented structure of the system and achieving efficient performance at the same time. The presented study is a practical lesson about the implementation of a scalable system oriented towards information dissemination. The key idea is to represent a large observable collection of objects by a repository that provides access to them both as individual CORBA objects and data records. The proper usage of this duality may have substantial influence on the overall system performance. The repository is equipped with a scalable notification mechanism built around a notification dispatcher and notification tree concepts. Fundamental features of the proposed solution are illustrated by a performance study and a representative application.

1 Introduction

Many existing information systems may be classified as information dissemination applications [4]. Those systems deliver information about changes of the interesting subset of data to the group of interested users. New approaches to a construction of dissemination systems, namely object-orientation and distribution, introduce a new problem of system scalability. So far, there are few attempts to build such systems using those modern technologies from scratch and there is no general answer to the scalability problem. One of the most crucial design decisions is the choice of a degree of an abstraction level of objects composing the system. For example, when building a CORBA-based system disseminating share prices, one should decide whether individual share prices will be represented as CORBA objects or not. Generally, this is a question how to map the objects of the system model into CORBA objects. Until now, there is a common conviction that a CORBA-based system built of a huge number of CORBA objects is inherently inefficient. The solution presented in this paper relaxes this limitation and proposes a template of a CORBA repository component with the incorporated light-weight mechanisms of notification, persistency and security.

This repository component combines and refines some ideas that recently appear in component oriented software environments such as Java Beans [12, 16], San Francisco Components [3], or CORBA Component proposed by Iona Ltd and others [15]. The major innovations are a dual form of access to repository entities that is by value or by CORBA references, and a scalable notification mechanism with built in smart proxies. Finally, the idea of dynamic attributes [7] has been exploited and three different types of the repository component have been proposed.

The structure of the paper is as follows. In Section 2 Active Badge next generation project, which induces the presented study is described. The repository component template is defined in Section 3. It is an observable component with dynamic attributes and a built-in searching engine. Changes of its attributes are propagated via notification mechanism. Implementation issues of this component are described in Section 4. The notification dispatcher concept used to built this mechanism is explained and a short description of the repository persistency and security functionality is presented.

Features and scalability of the proposed notification mechanisms are then analyzed in more details. This section includes also comparison of the proposed notification mechanism with CORBA Event Services. Next, in Section 5 performance evaluation study that concerns the investigated scalability is reported ad discussed. Section 6 presents basic application utilizing information gathered by ABng system. The paper ends with conclusions.

2 The Active Badge next generation project

The system, called *Active Badges*, was originally invented and developed at Olivetti Research Laboratory, in Cambridge, UK [6] in 1990-92. It uses hardware infrastructure whose key components are infra-red sensors, installed in fixed positions within a building, and infra-red emitters (*active badges*) that are worn by people or attached to equipment. Sensors are connected by a wired network which provides a communication path to the controlling device, called poller, and distributes low-voltage power. A poller is implemented as a PC or a workstation with a sensor control software active on it. An active badge periodically transmits an infra-red message containing a globally unique code (a badge identifier) using the defined data link layer protocol [2]. Messages are received and queued by sensors. A poller periodically polls sensors, and retrieves badge messages from sensor queues. Each badge message as well as an identifier of the sensor which received the message is forwarded to the software part of the Active Badges system. The software layer maintains a database that maps sensors to places in which sensors are installed and badges to users wearing these badges and to pieces of equipment which badges are attached to. Using these data the system can infer where users or pieces of equipment are currently located. The information about the current location of users and equipment is provided to various applications, such as presentation tools which display location data or applications which use location data to control users' environment. The software part of the original Active Badge system developed at ORL uses ANSAWare [1] distributed environment.

2.1 Goals of ABng project

The ABng project (Active Badges – next generation) aims at development of a new software layer of the Active Badge system that fulfills the following assumptions:

- is flexible and reconfigurable;
- separates the details of gathering of location data from the application layer;
- provides location data filtering;
- ensures privacy of location data and security;
- enables to build systems making a user's environment location-aware.

To satisfy the first requirement, ABng uses the modern component and object-oriented technology. The system is developed in CORBA-compliant environments: Orbix [8] and OrbixWeb [11]. It is based on the object model in which all logical and physical elements of the Active Badge system (users, locations, sensors, badges, etc.) are represented as CORBA objects.

The system has a layered architecture which hides details of gathering location data. This makes it possible to replace a localization method based on infra-red sensors and emitters by another one. In ABng location data are presented using abstract notions of *location* and *locatable* objects rather than in terms of sensors and badges. A location is a part of an environment obtained as a result of partition of the space according to an arbitrary, user-defined rule. Typically, an office space can be divided into buildings, floors, rooms, etc. A locatable is an object which can be observed by the system and whose location changes within the environment space can be monitored. A locatable can be a person or a piece of equipment, such as a computer, a printer or a book.

The basic ABng concept is *View* which is a collection of some location and locatable objects, i.e. it represents a part of the environment space and a subset of objects that can be localized within this part. The precision of localization of *View*'s locatables is equal to the size of locations belonging to the *View*. Within a system a number of *View* objects can exist, each of which can hold information

concerning current locations of users of equipment belonging to different groups and provided at different levels of abstraction with different precision.

The concepts of locations, locatables and *Views* are crucial for data filtering and protection of privacy of location data. Every application can individually decide which *View* and which locations or locatables, contained in the *View*, it is willing to observe. It can subscribe to interesting objects and, as a consequence, to receive the required data related to these objects. With every *View* existing in the system a list of users, who can access this *View*, is associated. Thus only these users have access to location data as well to other attributes of locations and locatables contained in the *View*.

The ABng incorporates development of the *Wonder Room* location-aware users' environment over the location system. This environment consists of a number of applications which control various elements of the users' equipment. Examples of such applications are redirection of phone calls to the currently nearest phone or setting parameters of various home appliances, such as air-conditioning, TV sets, VCRs, light, according to the preferences of users located in the neighborhood of these appliances, period of time, etc. Such applications may be used for *personalization* of user's equipment. Systems of this type are examples of, so called, ubiquitous computing [17].

2.2 Design considerations

After the analysis of the desired functionality of the ABng system many kinds of entities have been singled out which have to be represented in the software. These entities could be divided into two main categories:

- Closely related to Active Badge System configuration, such as *Sensor* and *Badge* on the lowest level, and *ABng_Location_Description* and *Badge_Holder* above it,
- Describing office environment in which Active Badge System was installed such as *User*, *Equipment*, *Location_Type*, *Location* and *View*, etc.

These entities do not only encapsulate their states but also possess more or less complex functionality. For instance, a request to play some sound could be sent to the badge or particular instance of equipment, such as air conditioning in the given room, could be requested to change its state. The last functionality is possible thanks to the integration with the infra-red controlling system. Generally, it was assumed that functionality linked with the given types of entities could evolve and be significantly extended, in the future.

Besides these numerous relationships between entities were grasped. A state of some entities depends or even is composed of the states of others. Thus to present the whole state of such an entity information from many other entities has to be gathered. It is for instance justified to separate description of particular part of location, such as room or floor, from entity encapsulating a set of sensors installed there. The sufficient reason for this is that description of a room or floor is universal while a set of sensors is ABng specific. Combining these two entities into one will make evolution or replacement of the ABng with other location system impossible.

Additionally, an entity should be immediately informed about the changes in states of entities it depends upon so as it can modify functionality of this entity and of the system. For instance, when a sensor is replaced or added to some room related *ABng_Location_Descriptions* and *Views* have to be informed, which in result will change processing of the sighting. Source of changes in states of entities can be:

1. a system administrator, when updating repositories with data describing ABng configuration and office environment – this changes are relatively rare and not bursty,
2. a movement of a locatable object – this changes are usually very often,
3. changes in the state of equipment – this changes may be often.

Such a change should be propagated not only within the system but should be further disseminated to interested observers. Thus the appropriate mechanism for managing lists of observers is necessary.

Because of all these reasons each of the singled out entities appears to be complex enough to justify its representation as a separate CORBA object. The result of such a decision is that there is a large

number of independent CORBA objects in the real ABng system with even moderate number of users.

3 Component template

The conclusions from the investigation of the previous system, which were applied during the system development are:

- General templates for an entity as well as a repository can and have to be designed,
- These templates should provide support for the implementation of a mechanism eliminating the overhead related to the representation of each entity as a separate CORBA object,
- A light-weight notification mechanism for repository clients has to be invented.

In the ABng three types of entities, and in the result three types of entity interfaces have been differentiated, with:

- **static attributes**: Most of ABng entities have a fixed and relatively small number of attributes. Such entities are accessed via interfaces, in which each entity attribute is represented by a corresponding IDL attribute.
- **dynamic attributes**: Another approach is to treat an entity as a collection of attributes of arbitrary types and to provide an access to them via an adequate interface. Such an interface offers a pair of access operations to set and retrieve the value of a single attribute in which an attribute is referred by its name and a value is decoded using the IDL *Any* type. The interface allows to retrieve all attributes as a list. This approach is an example of the application of the *Dynamic Attribute* design pattern [7]. This type of an interface is provided by entities which have many attributes or these attributes are different for individual entity instances. The ABng example of such an entity is the *Location_Description* object, which describes a piece of an office space, such as a floor, a room or a building. These real-world objects are inherently different and it is impossible to design a uniform set of attributes for them.
- **hybrid attributes**: This type of an interface explicitly defines these attributes which are common for all objects representing by entities. Additionally, it uses the *Dynamic Attribute* paradigm to provide an access to object-specific attributes. An example of an ABng object providing such an interface is *Equipment* whose instance describes a piece of office equipment. For all kinds of equipment a set of common attributes has been distinguished, such as a name, a vendor name, etc., which has been defined as explicit attributes.

For every type of interface a corresponding template has been designed. Below, the template for interfaces with static attributes is described in details as an example.

The template of an entity interface was defined as follows:

```
interface Entity: Entity_Observed,
  Entity_Commander {

  struct Description {
    Type1 attribute1;
    // ...
    TypeN attributeN;
  };

  typedef sequence<Description> Descriptions;

  struct Value_Description {
    Type1::Value_Description attribute1;
    // ...
    TypeN::Value_Description attributeN;
  };

  typedef sequence<Value_Description>
    Value_Descriptions;

  struct Pattern {
    boolean is_any;
    Type1::Pattern attribute1_pattern;
    // ...
    TypeN::Pattern attributeN_pattern;
  };

  readonly attribute Repository_Item_Id item_id;
  readonly attribute Description descr;
  readonly attribute Type1 attribute1;
  // ...
  readonly attribute TypeN attributeN;
}
```

The template defines a set of attributes: *attribute1*, ... ,*attributeN*. It also inherits from the *Entity_Comander* interface which defines its specific functionality.

Each entity possesses a unique identifier (*item_id*) inside a repository, which can be used by repository clients to refer to objects. This is an alternative to using object references for this purpose.

Besides, every entity inherits from the *Entity_Observed* interface which enables other objects to register their interest in changes of an entity state. When the state is changed the registered parties are informed about it.

The second uniform feature is the *descr* attribute of the *Description* type, which is a structure possessing fields corresponding attributes of a given entity. This structure is used to get the whole meaningful state of the entity in just one request. This feature was mainly designed in order to be used by the notification mechanism based on smart proxies, described in the next section. A smart proxy on the client side can retrieve the whole state of the entity when it is created and then serve a local request using cached data. It will also retrieve the whole state when cache is invalidated by the notification mechanism, which is described later on.

The next uniform element is a definition of the *Value_Description* structure. Like *Description* it has a field for every entity attribute but the type of this field is either the type of a corresponding attribute, providing it is not an object reference, or the *Value_Description* structure from the entity referenced by this attribute. The purpose of this approach is to enable to return the entity state as a set of already collected data without any references to the outside objects.

Finally, there is the *Pattern* structure which is built in a recursive way. It contains *Pattern* structures for simpler data types. The *is_any* field denotes if the *value field* is meaningful or not. The *Pattern* structure is used to specify searching criteria for the given entity type.

The templates for interfaces with dynamic or hybrid attributes (not presented here) are very similar to the above mentioned one. The difference is that in a dynamic interface the only attribute is a sequence of name/value pairs and there are two additional methods to set and retrieve a single value. In a interfaces with hybrid attributes occur both explicit attributes and a list of name/value pairs accompanied by access operations.

Objects built according to any of the three templates are stored in repositories which also possess a generic interface:

```
interface Entity_Rep : Entity_Rep_Observed {

  typedef sequence<Entity> Entities;
  readonly attribute Entities entity_list;

  Entity add(in Entity::Description data_record)
    raises(Duplicate_Data_Record);

  void remove(in Entity object_to_remove,
    in Entity::Description data_record)
    raises(Unknown_Object_Ref,
      Duplicate_Data_Record);

  void remove(in Entity object_to_remove)
    raises(Unknown_Object_Ref);

  Entities find(
   in Entity::Pattern data_record_pattern);

  Entity::Value_Descriptions find_values(
    in Entity::Pattern data_record_pattern);

  Entities find_and_attach(
   in Entity::Pattern data_record_pattern,
   in Observer obs,
   out Entity::Descriptions descritions);

  void update(in Entity object_to_update,
    in Entity::Description data_record_pattern)
    raises(Unknown_Object_Ref,
      Duplicate_Data_Record);
}
```

The operations of a repository are a follows.

- *add* – creates and adds a new entity to the repository. The initial state of the created object is determined by the contents of the *Description* structure passed as an argument. This method returns the object reference of the new entity.
- *update* – replace the state of the entity denoted by the reference with values stored in the *Description* structure.
- *remove* – remove an entity denoted by the reference.
- *find* – returns a collection of references of these entities which match the criteria specified in the *Pattern* structure passed to the operation.
- *find_values* – returns a collection of the states of the entities matching the given criteria. In other words, this operation returns the matching object by value rather than by reference.

- *find_and_attach* – works like the *find* method but additionally registers the observer (`obs`) for all returned entities in a repository. It also possesses an *out* parameter by which the sequence of entities descriptions is returned.

The entity repository interface inherits from the *Entity_Rep_Observed* interface, enabling registration of an interest in the arbitrary collection of entities. This facilitates registration of an observer in the large number of entities. A state of every entity contained in a repository is made persistent by the persistency mechanism used in the repository. The access to an entity is guarded by the security service.

4 Implementation of component facilities

Every ABng component offers mechanisms for asynchronous notification about changes of components attributes, for life cycle control, and security. They are examined below.

4.1 Light notification mechanism

A typical ABng application can use a lot of various information encapsulated by ABng objects. To obtain this information an application interacts with various ABng objects. To optimize these interactions, ABng implements a caching algorithm – Smart Proxy Layer (SPL), based on the smart proxy mechanism available in Orbix and OrbixWeb. When the application obtains an entity reference (for instance by executing the repository *find* method), the smart proxy of the entity object is instantiated within the application's address space and entity's descriptions is cached. When the application enquires about an attribute value, this value is retrieved from the cache and no remote call is performed.

If a value of an entity attribute changes, all proxies active in different applications have to be notified about this change. For this purpose, in ABng a notification mechanism based on the Observer [5] architectural design pattern (also known as Publisher/Subscriber) has been designed. Every proxy registers itself as an observer of the entity it represents. Each time, an attribute of the entity is updated, the proxy is notified and after that it marks its cache as invalid. The next application's query about an attribute value causes the proxy to contact the entity and retrieve the whole description. Registering smart proxies directly within an entity object would be very inefficient as for every smart proxy a corresponding proxy object in the server containing the entity would be created. To solve this problem the mechanism of notification dispatching is employed. This is depicted in Figure 1 and explained below.

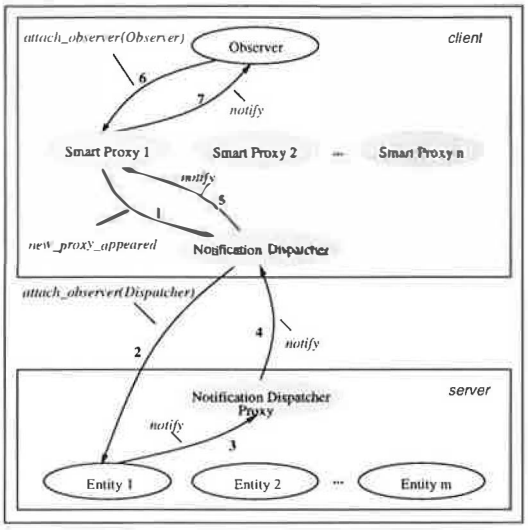


Figure 1: ABng notification mechanism

When a proxy representing the first entity from a given repository is instantiated within an application, an object, called *notification dispatcher* associated with the repository, is created. The dispatcher will represent all proxies associated with entities contained in the given repository and dispatch notification messages to the proxies. The smart proxy does not directly subscribe itself to the corresponding entity. Instead, it calls the notification dispatcher (arrow 1 in Figure 1). The dispatcher casts the smart proxy reference to the reference of an ordinary proxy and calls the real stub of the registration method (*attach_observer*) passing its own reference as an argument (the dispatcher has to provide the observer interface). This call results in a real remote invocation on the entity (arrow 2). If the dispatcher contacts the entity server for the first time, the dispatcher proxy is instantiated within the server at the same time. Within the repository the number of existing dispatcher proxies is equal to the number of applications which contain proxies observing

repository's entities. The entity stores a dispatcher reference (a pointer to a dispatcher proxy) in a registry of its observers.

When the state of the entity is changed the entity notifies all notification dispatchers via their proxies (arrows 3 and 4). On the application's side, the dispatcher obtains a reference of the updated entity, which, in fact, points to the local smart proxy. The dispatcher forwards notification to the smart proxy (arrow 5). Finally, the proxy marks its cache as invalid.

Beside smart proxies maintaining caches, inside a user's application there can be also ordinary objects which are interested in asynchronous notification about entity updates. This notification is also performed using the described mechanism. An application object, which wants to be notified about changes of an entity's state, calls the *attach_observer* operation of the interesting entity (arrow 6). This call, however, is not transmitted to the entity. It only affects a local registry of entity observers, which is maintained by the smart proxy. After the proxy is notified about entity update, it forwards this notification message to all entity observers contained within the application (arrow 7).

It should be noted that the above mechanism is completely transparent to the application. The application which is assumed to use this mechanisms has to be linked with the library containing smart proxies and dispatchers.

4.2 Persistency

States of entities have to survive rebooting of the system. Their persistency can be implemented using different approaches. However, the heavy and cumbersome mechanism could have a tremendous impact on efficiency and scalability of the system. In the ABng two versions of persistency mechanism are implemented:

- *File-based* – this primitive mechanism uses a separate file to store the serialized state of each repository. It is implemented as coarse-grained, which means that when one of the entities in the repository is changed then the whole state of the repository (all entities) is restored in the appropriate file.
- *Object Database Object Adapter (ODOA) [10]* – this sophisticated mechanism of achieving CORBA objects persistency uses an object database (ObjectStore [13]) to save separate objects as well as collections of objects. Each repository is a root for a collections of entities. However, when particular entity is changed only its state is updated in the database. The disadvantage of this mechanism is that a transaction has to be created. The ODOA provided by Iona is only single-threaded and always opens a heavy *update* transaction. The OOA used in the ABng was obtained as a specialization of the Object Database Adapter Framework [9]. Its special features enable multithreaded implementation of servers as well as instrumentation of ODOA, during the compilation of the program, with names of methods (together with interface names) requiring creation of *update* database transactions. In the other case the light *read-only* transaction is created.

The version of the persistency mechanism used in the given repository is determined during compilation (possibility of postponing this to the execution time is now investigated). There is no restriction that all repositories in the running system have to use the same persistency mechanism, each can adopt an adequate version of it.

4.3 Security

The access to entities is granted basing on the *View* level and thus it has to have effect on many other objects in other repositories associated with the given *View*. The authorization server connected with the *View Manager* automatically grants a user access to all data about locations and locatables of the *View*. This authorization data is replicated in caches within repositories. Therefore, when a user is denied access to the *View* or the contents of the *View* is changed (a location or a locatable is removed from the *View*) the authorization server informs repositories caches about this changes. The same notification mechanism as described in the previous section is employed here.

4.4 The scalability of notification mechanism

The light weight notification mechanism described in Section 4.1 seems to be promising for dissemination of information in a large community of clients distributed in the network. In this section the issue of its scalability is further analyzed.

The proposed solution of the notification has the following structural features:

- Notification dispatcher is a CORBA object which represents collection of smart proxies.
- The smart proxies are not CORBA objects and may be effectively notified using local method invocation call.
- The collection of repository entity smart proxies in the client space is represented in repository only by one notification dispatcher proxy. It saves a significant amount of memory and makes the repository sever occupied space independent on the number of entities in the collection and the global number of existing proxies.
- A client may be not aware that notification dispatcher is used. Its activity is completely transparent to the client even from the programming point of view.
- The number of the notification dispatcher proxies in the repository server is dependent only on the number of clients in the system which contain entity smart proxies.
- The proposed notification mechanism is selective, which means, that only this notification dispatchers are notified which have registered the smart proxies, corresponding to the changed entity in the repository.

It is necessary to point out that the existence of a notification dispatcher does not influence scalability of the system in terms of number of observer clients.

This last drawback could be overcome using replication. The client component with collection of smart proxies and notification dispatcher could be generalized as a notification component shown in Fig.2. The scalability with respect to the number of clients could be achieved by organizing the system into notification tree built of notification components. Each smart proxy has registered several notification dispatchers of the higher layer, etc. In the root node the repository of entities exists. In other nodes only smart proxies are present. The proposed architecture represents in fact a distributed collection of entities which could be highly available for large number of clients despite of geographic distribution.

The propagation of the repository entity update is marked for example in Fig.2. It is easy to see that the proposed solution has a similar scalability to notification based on multicast over IP communication protocols that in fact propagate messages down to a multicast tree. The advantage is that the notification tree does not require any multicast protocol support.

In context of this discussion it is necessary to ask about comparison of proposed solution with CORBA Event Services. There are some similarities and differences. The most important difference is that Event Service does not use a smart proxy concept so caching has to be solved in separate way. The notification component is similar to the event channel in the sense that it separates the repository as a source of events from the client. Further comparison is very much dependent on implementation details which are not defined by the OMG specification. For instance, in Iona's implementation based on a multicast protocol usage an events producer does not even know the number of notified consumers. This approach scales well but is based on the proprietary protocol and is very difficult to extend from LAN to WAN. To achieve selective notification it is necessary to define as many event channels as many sources of notification exist.

5 ABng system evaluation

The ABng software, implemented according to the design concepts presented in this paper, was subject to intensive testing in regards to its performance and scalability. The system was implemented in Orbix 2.2MT whereas clients, used in tests, were implemented in OrbixWeb 2.01. OrbixWeb 3.0 was not used as its mapping of a sequence of object references, when returned by a server is faulty: smart proxies are not created for returned references. This results in incorrect operation of the Smart Proxy Layer proposed in this paper.

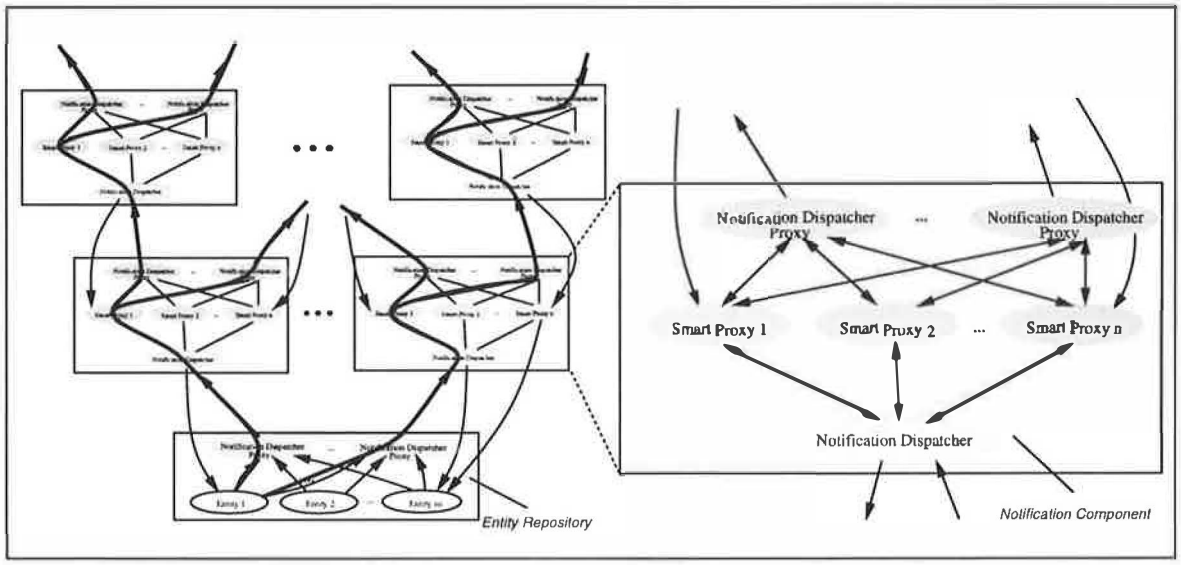


Figure 2: Notification Tree

Performance results presented in this section were obtained in the environment consisting of 15 Sun Ultra workstations and a Sun Enterprise 3000 server, connected by 2 Ethernet 10Mb switches. ABng system components were running on the server. All other programs were executed on separate workstations, so all CORBA invocations went through the network. One of the system component was chosen for the test. However, results are representative of all of them as they were implemented using the same C++ template. All tests were repeated 100 times and average values were calculated. The workstations were used for usual activities during the test, but were rather slightly loaded.

Performance tests were carried out according to typical scenarios occurring in majority of ABng applications i.e: retrieving of the current state of the entities in an application bootstrap and scalability of notification mechanism when numbers of applications observing changes in entities increase. Improvements of implementation of these activities proposed in the paper – SPL and notification were also evaluated.

5.1 Costs of accessing repository entities

There are two different ways of accessing entities in repository: by their values, using *find_value* and by their references, using *find*. In order to obtain values of entities, when using *find*, subsequent *get_descr* methods have to be called. Time spent in these calls executed in the OrbixWeb applet when numbers of entities in repository increase is presented in Fig. 3 and Fig. 4. All these calls were invoked with a *pattern* matching all entities in the repository. Results show that accessing entities by references is by few magnitudes more costly than accessing them by value. Thus access by references, in spite of its many superior features, has to be used carefully and often combined with access by value, as in the application presented in Section 6.

These figures present also performance of 2 subsequent invocations of the *find* method when SPL is used. The execution time of the second *find* one is obviously almost neglectable as it happens locally. Additionally, the first call with SPL is about 40% more costly than combined *find* without SPL and *get_descr* calls. This difference is caused by the SPL construction time.

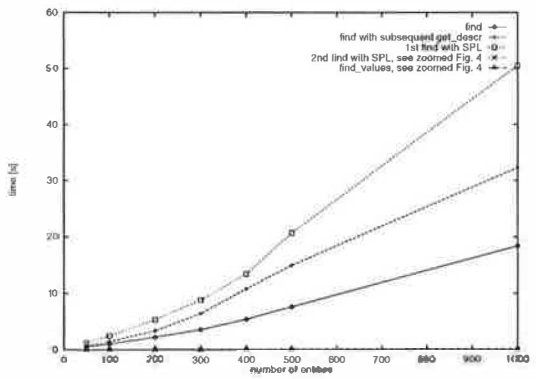


Figure 3: Performance comparison of different ways of accessing states of repository entities

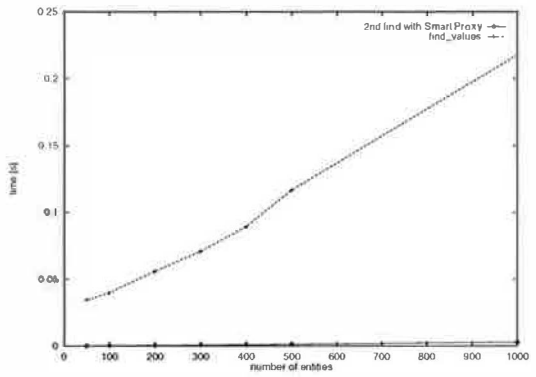


Figure 4: Performance comparison of different ways of accessing states of repository entities, zoomed

5.2 Analysis of SPL construction cost

The process of constructing SPL was divided into four basic steps:

1. acquiring of entity references by executing the *find* method,
2. creating of smart proxy for each of the acquired reference,
3. registering of the notification dispatcher for each of the references,
4. retrieving of entity descriptions by executing the `get_desc` method for each smart proxy.

All of these steps, except the second one, include remote CORBA invocations. The first step includes one remote invocation, the third and fourth ones include as many remote invocations as many are acquired references. In Fig. 5 the total SPL construction time as well times sent in individual steps are presented.

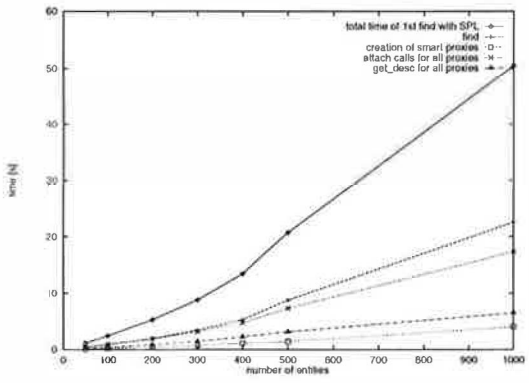


Figure 5: Time consumption by basic steps of SPL construction

These results show that majority of the time is used for remote invocation. Most of these invocations could be canceled by usage of the *find_and_attach* repository method, which eliminates necessity of remote calls from the third and fourth steps. This reduces the SPL construction time by 45%. However, this approach additionally requires construction of a smart proxy for the repository. In this smart proxy a *find* call is replaced by a *find_and_attach* call, with observer (parameter *obs*) initiated to the notification dispatcher. The returned sequence of entities values (out parameter *descriptions*) is used for filling smart proxy caches.

5.3 Notification time

The method used to notify observers registered in the repository about its changes are asynchronic *oneway* operations invoked successively on the observers. The impact of growing number of entity observers on the notification time is presented in Fig. 6.

The obtained results show that the time of executing *update* on an entity is almost independent of a number of its observers. The total time of informing all observers however increases by roughly 3 [ms] for each additional observer. It means that in one second only 330 successive notification calls can

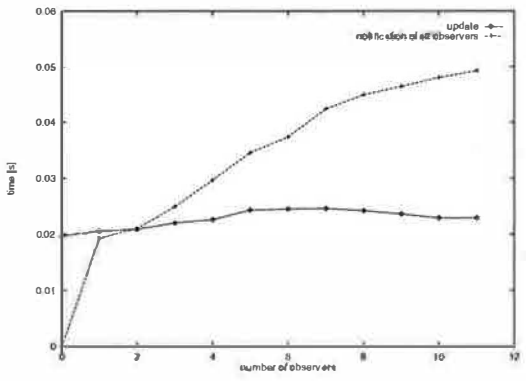


Figure 6: Impact of growing number of observers on entity update and notification times

be executed (obviously this number depends on the performance of a computer server). Thus, in the case of the used hardware, the product of a number of observers (`Obs`) and a number of events per second (`Evn`) could be at most 330:

```
Obs * Evn <= 330
```

In the case of the ABng system only the second source of events from these distinguished in Section 2.2, namely changes in location, can cause intensive stream of events. Each badge generates a new sighting every 10 [s], however statistically less than 4% of them carry meaningful information (every 4 minutes) i.e.: changes in location or clicking on one of badge buttons. Only such events are reported further to observing applications. By applying the formula to these figures we can expect that the system will scale up to 500 badges and more then one hundred observer applications, when an adequately fast server computer is used. Moreover, filtering of events by the usage of the *View* concept, presented earlier in this paper, further reduces the stream of events.

To scale the system additionally it is necessary to apply: some multicast protocol, the notification tree proposed in this paper or a combination of these two approaches. In the case when a multicast protocol is used, for instance by the usage of the Iona's implementation of the Event Service, a number of observers in the formula `Obs` is equal to 1 and a number of meaningful events can reach 330 per second. By adding the notification component to the system this figure may be scaled further. The only disadvantage of this approach is a delay introduced by the notification component in the delivery of events to its observers. Moreover, the notification component is necessary when the system has to be extended geographically over WAN, which is very rarely configured for multicast.

6 A representative ABng application

The basic ABng application is the ABng viewer, called Jabba. The primary function of the viewer is to present a list of users with their current locations (Figure 7). Similarly, a list of equipment can be displayed. For every object a number of its attributes are presented (e.g. a user name, a user address. a location name, location phone numbers, etc). To perform these tasks the viewer has to collect a lot of information which is distributed among various repository servers. Additionally, information presented to the user has to be refreshed after any of the relevant repository object changes one of its attributes. A change may concern the current location of a user or a piece of equipment or other attributes, such as a user address or a location description.

To work efficiently the viewer has to maintain a local copy of relevant information, i.e. to cache values of object attributes and to update values in caches after their originals are modified. In the first, ANSA-based version of the Active Badge location system, the viewer, called *xab* was a very sophisticated and huge application and the most of the *xab* code was related to maintaining caches. In ABng viewer caching is implemented by the smart proxy layer. This has three major advantages:

- The code related to caching is completely separated from the application code. The application is not responsible for updates of the local copies of information.
- The application code is not aware that any caching algorithm is performed. It is completely transparent for the application. When the application wants to obtain an attribute value (e.g. in order to redisplay it on the screen), when it got informed of the change, it just calls the operation of the remote object which holds that attribute. However, the call does not come out from the client application's

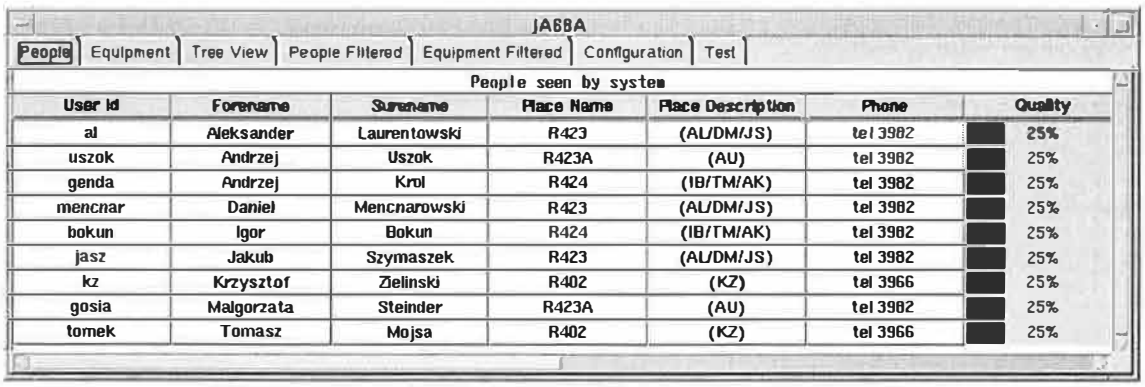


Figure 7: ABng viewer – a list of locatable users

address space. It is caught by the smart proxy layer and served locally.

- The smart proxy layer is universal and it can be reused in any ABng application.

In the bootstrap of Jabba the hybrid approach to retrieving data from repositories was employed. First, all entities are retrieved by value, so their attributes can be very quickly displayed. In the background references of these entities are acquired and SPL is built. It takes considerable amount of time, however it is transparent for the user. When the SPL is built it very efficiently serves Jabba functionality.

7 Conclusion

Construction of scalable components in CORBA requires solution of well known trade-off between a space and a simplicity of navigation in a large collection of objects on the one hand and a system time of reaction which is a major scalability factor on the other hand. The access by CORBA references provides conceptually clear and elegant model of access to objects in a distributed system but when their number increases it induces not acceptable access time. On the contrary access by value is much faster but is loosing the ability of easy navigation in a distributed system. So the solution is to build hybrid components which combine the both proposed in this paper mechanisms. It is up to a programmer to use them correctly. Some hints in this matter are performance tests presented in the paper.

The similar conclusion could be drawn in respect to the notification mechanism proposed in this paper. It works very well after the initial phase when the notification tree and smart proxies are already established but this phase takes a substantial amount of time. So for a client which needs fast response time the initial value of entities should be get by value at first place and the notification tree should be constructed in parallel for future accesses and notification.

The design and implementation solutions presented in this article proved its correctness and scalability in the working ABng system.

The presented repository component may be further enhanced by using the POA [14] approach, that provides new standard scalability mechanisms. It is our intention to follow in this direction.

8 Acknowledgments

This work is supported by Olivetti-Oracle Research Laboratory, Cambridge, UK.

References

[1] *ANSAware 4.0 – Application Programmer's Manual*, APM Ltd., Cambridge UK (1992).

[2] F. Bennett, A. Harter, *Low bandwidth infra-red networks and protocols for mobile communicating devices*, Technical Report 93.5, Olivetti Research Laboratory, Cambridge, UK (1993).

[3] K. Boher, *Middleware Isolates Business Logic*, Object Magazin, 11 (1997).

[4] M. Franklin, S. Zdonik, "A Framework for Scalable Dissemination-Based Systems", *Proceedings of OOPSLA '97* (1997) p. 94–105.

[5] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns*, Addison-Wesley (1994).

[6] A. Harter, A. Hopper, *A distributed location system for the active office*, IEEE Network, Special Issue on Distributed Systems for Telecommunications, 8(1), January (1994).

[7] T. Mowbray, R. Malveau, *CORBA Design Patterns*, John Wiley and Sons, Inc. (1997).

[8] Iona Technologies Ltd., *Orbix 2.1 Programming Guide* (1996).

[9] Iona Technologies Ltd., *Orbix Database Adapter Framework – White paper* (1997).

[10] Iona Technologies Ltd., *Orbix+ObjectStore Adapter Programming Guide* (1997).

[11] Iona Technologies Ltd., *OrbixWeb 3.0 Programming Guide* (1997).

[12] R. Orfali, D. Harkey, J. Edwards, *The Essential Distributed Objects Survival Guide*, John Wiley and Sons, Inc. (1996).

[13] Object Design, Inc., *ObjectStore C++ API User Guide, Release 4.0.1* (1996).

[14] Object Management Group, *Specification of the Portable Object Adapter (POA)*, OMG Document orbos/97-05-15 (1997).

[15] Object Management Group, *CORBA Components, Joint Initial Submission by IONA Technologies et al.*, OMG Document orbos/97-11-24 (1997).

[16] P. Sridharan, *Java Beans, Developer's Resources*, Printice Hall (1997).

[17] M. Weiser, *Some computer science issues in ubiquitous computing*, Communications of the ACM, 6 (1993) p. 75–84.

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits:

- Free subscription to *;login:*, the Association's magazine, published eight times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Javan and C++, book and software reviews, summaries of sessions at USENIX conferences, Snitch Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, via the USENIX Online Library on the World Wide Web.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as object-oriented technologies, security, operating systems, electronic commerce, and NT - as many as ten technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- PGP Key Signing Service (available at conferences).
- Discount on BSDI, Inc. products.
- Discount on the five volume set of 4.4BSD manuals plus CD-ROM published by O'Reilly & Associates, Inc. and USENIX.
- Discount on all publications and software from Prime Time Freeware.
- 20% discount on all titles from O'Reilly & Associates.
- Savings (10-20%) on selected titles from McGraw-Hill, The MIT Press, Morgan Kaufmann Publishers, Sage Science Press, and John Wiley & Sons.
- Special subscription rate for *The Linux Journal* and *The Perl Journal.*
- The right to vote on matters affecting the Association, its bylaws, election of its directors and officers.

Supporting Members of the USENIX Association:

Adobe Systems Inc.
Advanced Resources
ANDATACO
Apunix Computer Services
Auspex Systems, Inc.
Boeing Commercial
Crosswind Technologies, Inc.
Digital Equipment Corporation
Earthlink Network, Inc.
Invincible Technologies
Lucent Technologies, Bell Labs
Motorola Research & Development
MTI Technology Corporation
Nimrod AS
O'Reilly & Associates
Sun Microsystems, Inc.
Tandem Computers, Inc.
UUNET Technologies, Inc.

Sage Supporting Members:

Atlantic Systems Group
Digital Equipment Corporation
ESM Services, Inc.
Global Networking and Computing, Inc.
Great Circle Associates
OnLine Staffing
O'Reilly & Associates
Sprint Paranet
Texas Instruments, Inc.
TransQuest Technologies, Inc.
UNIX Guru Universe

For further information about membership, conferences or publications, contact: USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA. Phone: 510-528-8649. Fax: 510-548-5738. Email: *office@usenix.org.*
URL: *http://www.usenix.org.*

ISBN 1-880446-93-6